

SUBMISSION OPERATING SYSTEM

# Tóm tắt lý thuyết và câu hỏi chương 1-10

---



TP. HCM 5/2020

PHÂN CHIA CÔNG VIỆC NHÓM

Nhóm lớp (tổ)	MSSV	Họ và tên	Công việc
L04	1812496	Nguyễn Hữu Hưng	Chương 1: Giới thiệu
L04	1810340	Võ Hoàng Hải Nam	Chương 2: Cấu trúc Hệ điều hành
L04	1813910	Đặng Ngọc Tâm	Chương 3: Process
L04	1810299	Phan Quốc Long	Chương 4: Thread và thực thi đồng thời
L01	1810987	Đỗ Việt Vân Khanh	Chương 5: Đồng bộ và giải quyết tranh chấp
L02	1812501	Phạm Duy Hưng	Chương 6: Các ví dụ về đồng bộ
L04	1810648	Huỳnh Thị Uyên	Chương 7: Định thời CPU
L04	1811661	Bùi Tấn Danh	Chương 8: Deadlocks
L04	1811015	Lê Phương Khuê	Chương 9: Bộ nhớ chính
L05	1812327	La Quốc Nhựt Huân	Chương 10: Bộ nhớ ảo

# MỤC LỤC

## SUBMISSION OPERATING SYSTEM

Tóm tắt lý thuyết và câu hỏi chương 1-10

0

## CHƯƠNG 1: GIỚI THIỆU

1. HỆ ĐIỀU HÀNH LÀM NHỮNG CÔNG VIỆC NÀO	11
1.1 Cách nhìn người dùng	11
1.2 Cách nhìn hệ thống	11
1.3 Định nghĩa hệ điều hành	11
2. TỔ CHỨC HỆ THỐNG MÁY TÍNH	12
2.1 Interrupts	12
2.2 Cấu trúc lưu trữ	14
2.3 Cấu trúc I/O	15
3. KIẾN TRÚC HỆ THỐNG MÁY TÍNH	15
3.1 Hệ thống đa xử lý	15
3.2 Hệ thống cụm	17
4. CÁC THAO TÁC CỦA HỆ ĐIỀU HÀNH	18
4.1 Đa chương trình và đa nhiệm	19
4.2 Hoạt động ở chế độ kép và đa chế độ	19
4.3 Hẹn giờ	20
5. QUẢN LÝ TÀI NGUYÊN	20
5.1 Quản lý quy trình	20
5.2 Quản lý bộ nhớ	21
5.3 Quản lý hệ thống tệp	21
5.4 Quản lý lưu trữ lớn	22
5.5 Quản lý cache	22
5.6 Quản lý hệ thống I/O	23
6. TÍNH AN TOÀN VÀ BẢO MẬT	23
7. ẢO HÓA	24
8. HỆ THỐNG PHÂN TÁN	24
9. CẤU TRÚC DỮ LIỆU KERNEL	25
9.1 Lists, Stacks, and Queue	25
9.2 Trees	26
9.3 Hash Functions and Maps	27
9.4 Bitmaps	27
10. CÁC MÔI TRƯỜNG MÁY TÍNH	27

10.1	Máy tính truyền thống	27
10.2	Mô hình Di động	28
10.3	Mô hình client-server	28
10.4	Mô hình Peer-to-Peer	28
10.5	Mô hình đám mây	29
10.6	Hệ thống nhúng thời gian thực	30
11.	HOẠT ĐỘNG MIỄN PHÍ VÀ NGUỒN MỞ	30
12.	TỔNG KẾT	30
13.	QUESTIONS	31
<b>CHƯƠNG 2: CẤU TRÚC CỦA HỆ ĐIỀU HÀNH</b>		<b>34</b>
1.	DỊCH VỤ CỦA HỆ ĐIỀU HÀNH	<b>Error! Bookmark not defined.</b>
2.	GIAO DIỆN NGƯỜI DÙNG CỦA HỆ ĐIỀU HÀNH	35
2.1	Command Interpreter (CI)	35
2.2	Graphical User Interfaces (GUI)	36
2.3	Choice of interface	36
3.	LỜI GỌI HỆ THỐNG (SYSTEM CALL)	37
4.	PHÂN LOẠI SYSTEM CALL	38
4.1	Quản lý tiến trình (process control)	38
4.2	Quản lý tệp (file manipulation)	38
4.3	Quản lý thiết bị (device manipulation)	38
4.4	Duy trì thông tin (information maintenance)	39
4.5	Giao tiếp (communication)	39
4.6	Bảo vệ (protection)	39
5.	CHƯƠNG TRÌNH HỆ THỐNG	39
6.	THIẾT KẾ VÀ TRIỂN KHAI HỆ ĐIỀU HÀNH	40
6.1	Mục tiêu thiết kế	40
6.2	Chiến lược và cơ chế	40
6.3	Hiện thực	40
7.	CẤU TRÚC CỦA HỆ ĐIỀU HÀNH	40
7.1	Kiến trúc đơn giản	41
7.2	Phương pháp phân lớp	41
7.3	Microkernel	41
7.4	Modules	41
7.5	Hybrid Systems	42
8.	GỠ LỖI HỆ ĐIỀU HÀNH	42

8.1 Phân tích lỗi	42
8.2 Điều chỉnh hiệu suất	42
8.3 DTrace	43
<b>9. OPERATING-SYSTEM GENERATION</b>	<b>43</b>
<b>10. SYSTEM BOOT</b>	<b>43</b>
<b>11. CÂU HỎI</b>	<b>44</b>
<b>CHƯƠNG 3: PROCESS</b>	<b>47</b>
3.1 Khái niệm process	48
3.1.1 Process	48
3.1.2 Trạng thái process	49
3.1.3 Process control block	50
3.2 Process Scheduling	50
3.2.1 Scheduling Queues	50
3.2.2 CPU Scheduling	51
3.2.3 Context Switch	51
3.3 Các hoạt động trên process	51
3.3.1 Tạo process	51
3.3.2 Kết thúc process	52
3.4 Giao tiếp giữa các process	53
3.5 Giao tiếp giữa các process trong hệ thống sử dụng vùng nhớ chia sẻ	53
3.6 Giao tiếp giữa các process trong hệ thống truyền thông điệp	55
3.6.1 Naming	55
3.6.2 Bất đồng bộ	56
3.6.3 Buffering	56
3.7 Ví dụ hệ thống giao tiếp process	57
3.7.1 POSIX Shared Memory	57
3.7.2 Mach Message Passing	57
3.7.3 Window	57
3.7.4 Pipes	59
3.7.4.1 Ordinary pipes	59
3.7.4.2 Named pipe	59
3.8 Giao tiếp giữa hệ thống Client-Server	60
3.8.1 Sockets	61
3.8.2 Thủ tục lời gọi từ xa	62
3.9 Câu hỏi trắc nghiệm	62

<b>CHƯƠNG 4: LUỒNG &amp; THỰC THI ĐỒNG THỜI</b>	64
1. Tổng quan	64
1.1 Tại sao nên đa luồng (multithreaded)?	65
1.2 Lợi ích của multithreaded	65
2. Lập trình xử lý đa lõi (Multicore Programming)	65
2.1 Các vấn đề của việc lập trình trên hệ thống đa lõi	65
2.2 Định luật Amdahl	66
2.3 Các loại xử lý song song	66
2.3.1 Xử lý song song về dữ liệu (Data parallelism)	67
2.3.2 Xử lý song song về nhiệm vụ (Task parallelism)	67
3. Các mô hình xử lý đa luồng	67
3.1 Mô hình Many-to-One	68
3.2 Mô hình One-to-One	68
3.3 Mô hình Many-to-Many	69
4. Các thư viện luồng (Thread Libraries)	69
4.1 Pthreads	69
4.2 Windows Threads	69
4.3 Java Threads	70
5. Luồng ngầm (Implicit Threading)	70
5.1 Thread Pools	70
5.2 Fork-Join	71
5.3 OpenMP	71
5.4 Grand Central Dispatch (GCD)	71
5.5 Intel Threading Building Blocks (TBB)	71
6. Các vấn đề về thread	72
6.1 Ngữ nghĩa của fork() và exec()	72
6.2 Xử lý tín hiệu (Signal handling)	72
6.3 Hủy thread (Thread cancellation)	72
6.4 Lưu trữ cục bộ thread (Thread-local storage TLS)	72
6.5 Kích hoạt định thời (Scheduler Activations)	73
7. Ví dụ của hệ điều hành	73
7.1 Windows Threads	73
7.2 Linux Threads	74
8. Câu hỏi	75
<b>CHƯƠNG 5: ĐỒNG BỘ VÀ GIẢI QUYẾT TRANH CHẤP</b>	77

1. Đặt vấn đề	77
1.1. Race Conditions	78
1.2. Critical sections	78
1.3. Race Conditions trong Critical Sections	78
2. Vấn đề vùng tranh chấp (Critical Section Problem)	79
3. Các bài toán đồng bộ	80
3.1. Bài toán Producer/Consumer ( Bounded buffer)	80
3.1.1. Mô tả	80
3.1.2. Giải pháp	80
3.2. Bài toán "Bữa tối của các triết gia" ("Dining Philosophers")	81
3.2.1. Mô tả	81
3.2.2. Giải pháp	83
3.3. Bài toán Readers-Writers	83
3.3.1. Mô tả	83
3.3.2. Giải pháp	84
4. Phân loại giải pháp cho loại trừ tương hỗ	85
5. Giải thuật Peterson	87
5.1. Giải thuật Peterson cho 2 process	87
5.2. Tính đúng đắn	88
5.3. Đánh giá	88
6. Lệnh TestAndSet	89
7. Mutex lock	91
8. Semaphore	92
9. Monitor	93
10. Deadlock	94
11. Câu hỏi trắc nghiệm	94
<b>CHƯƠNG 6: CÁC VÍ DỤ VỀ ĐỒNG BỘ</b>	97
1 Các vấn đề kinh điển của đồng bộ	97
1.1. The bounded-buffer problem (Producer-consumer problem)	97
1.2. The Readers-Writers Problem	97
1.3. The Dining-Philosophers Problem (Vấn đề về bữa tối của các triết gia)	99
1.3.1 Giải pháp Semaphore	99
1.3.2 Monitor Solution (Giải pháp giám sát)	100
2. Đồng bộ trong Kernel.	101
2.1 Đồng bộ trong Windows:	101

2.2 Đồng bộ trong Linux.	102
3 Đồng bộ POSIX (POSIX Synchronization):	103
3.1 POSIX Mutex Locks:	103
3.2 POSIX Semaphores.	103
3.2.1 POSIX Named Semaphores.	103
3.2.2 POSIX Unnamed Semaphores(Semaphores Posix không được đặt tên):	104
4 Đồng bộ trong Java:	105
4.1 Java Monitors (giám sát)	105
4.2 Reentrant Locks	107
4.3 Semaphore	108
4.4 Biến điều kiện	108
5 Các cách tiếp cận thay thế:	109
5.1 Transactional memory (Bộ nhớ giao tiếp)	109
5.2 OpenMP	110
5.3 Functional Programming Languages (Ngôn ngữ lập trình hàm)	110
6 Summary (Tóm tắt)	111
7. Câu hỏi trắc nghiệm	111
<b>CHƯƠNG 7: ĐỊNH THỜI CPU</b>	114
1. CÁC KHÁI NIỆM CƠ BẢN	114
1.1 CPU-I/O Burst Cycle	114
1.2 CPU Scheduler	114
1.3 Định thời nhường	115
1.4 Dispatcher	115
2. TIÊU CHÍ ĐỊNH THỜI	116
3. GIẢI THUẬT ĐỊNH THỜI	116
3. 1 First-Come, First-Serve	116
3.2 Shortest-Job-First (SJF)	117
3.3 Shortest-Remaining-Time-First (SRTF)	118
3.4 Round Robin (RR)	118
3.5 Priority Scheduling	120
3.6 Multilevel Queue	121
3.7 Multilevel Feedback Queue	121
4. ĐỊNH THỜI THREAD	122
5. ĐỊNH THỜI CHO HỆ THỐNG MULTI-PROCESSOR	122
5.1 Tiếp cận định thời cho hệ thống multi-processor	123

5.2 Multicore Processor	123
5.3 Cân bằng tải	124
5.4 Processor Affinity	125
6. CÂU HỎI TRẮC NGHIỆM	125
<b>CHƯƠNG 8: DEADLOCKS</b>	130
I. MÔ HÌNH HỆ THỐNG (SYSTEM MODEL).	130
II. ĐẶC ĐIỂM DEADLOCK (DEADLOCK CHARACTERIZATION).	130
1. Những điều kiện cần thiết gây ra deadlock (Necessary Conditions).	130
2. Đồ thị cấp phát tài nguyên (Resource – Allocation Graph).	131
III. CÁC PHƯƠNG PHÁP XỬ LÝ DEADLOCK (METHODS FOR HANDLING DEADLOCKS).	132
IV. NGĂN CHẶN DEADLOCK (DEADLOCK PREVENTION).	132
1. Ngăn Mutual Exclusion.	132
2. Ngăn Hold and Wait.	132
3. Ngăn No preemption.	133
4. Ngăn Circular Wait.	133
V. TRÁNH DEADLOCK (DEADLOCK AVOIDANCE).	133
1. Trạng thái an toàn (Safe State).	133
2. Giải thuật đồ thị cấp phát tài nguyên (Resource – Allocation Graph Algorithm).	134
3. Giải thuật của Banker (Banker's Algorithm).	135
a. Giải thuật phát hiện an toàn (Safety Algorithm).	136
b. Giải thuật cấp phát tài nguyên (Resource-Request Algorithm).	136
VI. PHÁT HIỆN DEADLOCK (DEADLOCK DETECTION).	137
1. Mỗi loại tài nguyên chỉ có một thực thể.	137
2. Mỗi loại tài nguyên có nhiều thực thể.	138
3. Giải thuật phát hiện deadlock (Detection – Algorithm Usage).	138
VII. PHỤC HỒI DEADLOCK (RECOVERY FROM DEADLOCK).	139
1. Phục hồi khỏi deadlock bằng cách chấm dứt tiến trình (Process Termination).	139
2. Phục hồi khỏi deadlock bằng cách lấy lại tài nguyên (Resource Preemption).	139
VIII. CÂU HỎI TRẮC NGHIỆM.	140
<b>CHƯƠNG 9: BỘ NHỚ CHÍNH</b>	141
I. Phần cứng cơ bản:	141
1. Liên kết địa chỉ (Address Binding):	141

2. Không gian địa chỉ vật lý logic (Logical Versus Physical Address Space)	141
3. Nạp động (Dynamic loading):	142
II.Phân bổ bộ nhớ liên tục (Contiguous Memory Allocation)	142
1. Bảo vệ bộ nhớ (Memory Protection):	142
2. Cấp phát vùng nhớ (Memory Allocation):	142
3. Phân mảnh	143
III. Phân trang (Paging)	143
1. Phương pháp cơ bản:	143
2. Hỗ trợ phần cứng	143
3. Thời gian truy xuất hiệu dụng (Effective memory-access time):	144
4. Bảo vệ	144
5. Chia sẻ trang (Shared Page):	144
IV.Tổ chức bảng trang	145
1. Cấu trúc bảng phân trang theo kiểu kế thừa (Phân trang 2 cấp) - Hierarchical paging	145
1.1 Bảng phân trang 2 mức:	145
1.2 Bảng phân trang 3 mức:	146
2. Cấu trúc bảng phân trang theo kiểu nghịch đảo – Inverted page tables:	146
3. Cấu trúc bảng phân trang theo kiểu dùng hash function- Hashed page tables	147
4. Oracle SPARC Solaris	147
V. Cơ chế Swapping:	147
1. Tiêu chuẩn swapping	147
2. Swapping với cơ chế phân trang:	148
3. Swapping trên hệ thống di động:	148
VI. Question:	148
<b>CHƯƠNG 10: BỘ NHỚ ẢO</b>	151
1. Dẫn nhập	152
2. Phân trang theo nhu cầu:	153
2.1 Khái niệm cơ bản:	153
2.2 Danh sách khung trống:	156
3. Copy-on-write:	157
4. Thay trang	158
4.1 khái niệm thay thế trang:	158
4.2 Thay thế trang kiểu FIFO:	159
4.3 Thay thế trang tối ưu:	160

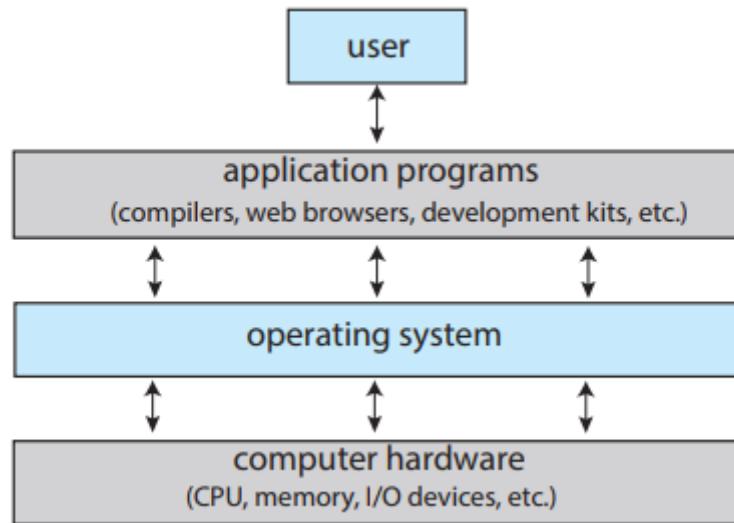
4.4 Thay trang kiểu LRU:	161
4.5 Thay trang xấp xỉ LRU	162
4.5.1 Additional-Reference-Bits	162
4.5.2 Second-Chance	162
4.5.3 Enhanced Second-Chance	163
4.6 Counting-Based	163
4.7 Page-Buffering	164
4.8 Ứng dụng và thay trang	164
5 Allocation of Frames	165
5.1 Cấp phát số lượng khung nhỏ nhất	165
5.2 Giải thuật cấp phát khung	165
5.3 Phân bổ cục bộ vs toàn cục	166
5.4 Non-Uniform Memory Access (NUMA)	167
6 Thrashing	168
6.1 Nguyên nhân	168
6.2 Mô hình working-set	170
6.3 Tần suất lỗi trang	171
7 Nén bộ nhớ	172
8 Phân bổ vùng nhớ kernel	173
8.1 Hệ thống Buddy	173
8.2 Phân bổ Slab	174
9 Những thứ liên quan khác	175
9.1 Chuẩn bị phân trang	175
9.2 Kích thước trang	175
9.3 TLB Reach	176
9.4 I/O Interlock	176
9.5 Cấu trúc chương trình	177
10 Ví dụ thực tiễn	177
10.1 Linux	177
10.2 Windows	178
10.3 Solaris	179
11 Tổng kết	180
12 Câu hỏi	182

# CHƯƠNG 1: GIỚI THIỆU

## 1. HỆ ĐIỀU HÀNH LÀM NHỮNG CÔNG VIỆC NÀO

### 1.1 Cách nhìn người dùng

Giao diện người dùng của máy tính thay đổi tùy theo giao diện được sử dụng.



### 1.2 Cách nhìn hệ thống

Theo quan điểm của máy tính, hệ điều hành là chương trình nhiều nhất liên quan mật thiết với phần cứng.

Một cái nhìn hơi khác về một hệ điều hành nhấn mạnh sự cần thiết phải kiểm soát các thiết bị I / O khác nhau và các chương trình người dùng. Một hệ điều hành là một chương trình điều khiển. Một chương trình điều khiển quản lý việc thực hiện các chương trình người dùng để ngăn chặn lỗi và sử dụng máy tính không đúng cách. Nó đặc biệt quan tâm với hoạt động và điều khiển của các thiết bị I / O.

### 1.3 Định nghĩa hệ điều hành

Làm thế nào, sau đó, chúng ta có thể định nghĩa một hệ điều hành là gì? Nói chung, chúng tôi có không có định nghĩa hoàn toàn đầy đủ của một hệ điều hành. Hệ điều hành tồn tại bởi vì chúng cung cấp một cách hợp lý để giải quyết vấn đề tạo một hệ thống máy tính có thể sử dụng. Mục tiêu cơ bản của hệ thống máy tính là để thực hiện các chương trình và để giải quyết vấn đề người dùng

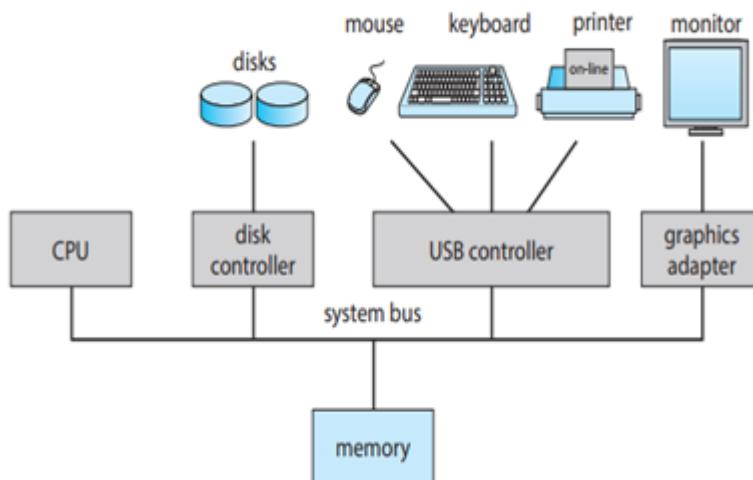
dễ dàng hơn. Máy vi tính phần cứng được xây dựng hướng tới mục tiêu này. Vì phần cứng đơn thuần là không đặc biệt dễ sử dụng, các chương trình ứng dụng được phát triển. Những chương trình này yêu cầu một số hoạt động chung nhất định, chẳng hạn như các hoạt động điều khiển các thiết bị I / O. Các chức năng phổ biến của kiểm soát và phân bổ tài nguyên sau đó được mang lại cùng nhau thành một phần mềm: hệ điều hành.

## 2. TỔ CHỨC HỆ THỐNG MÁY TÍNH

Tổ chức hệ thống máy tính :

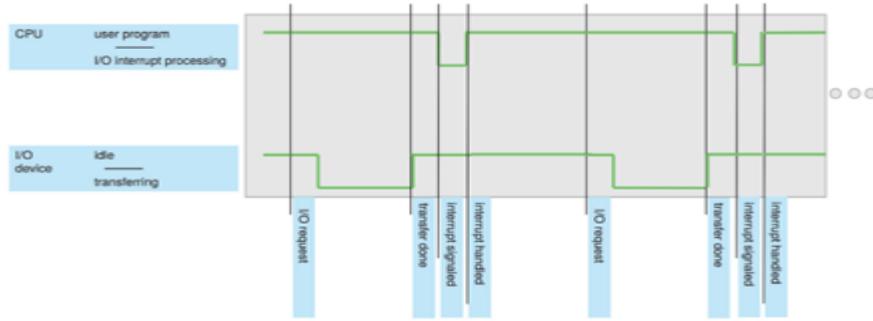
Một hoặc nhiều CPU, bộ điều khiển thiết bị kết nối thông qua bus chung cung cấp quyền truy cập vào bộ nhớ dùng chung.

Thực thi đồng thời CPU và thiết bị cạnh tranh cho chu kỳ bộ nhớ.



### 2.1 Interrupts

- Bộ điều khiển thiết bị thông báo cho CPU rằng nó đã kết thúc hoạt động bằng cách tăng ngắt hoặc CPU phải thực hiện một cuộc bỏ phiếu để hoàn thành I / O (có thể lãng phí một số lượng lớn chu kỳ CPU). Nói chung, chuyển điều khiển sang thói quen dịch vụ ngắt, thông qua vectơ ngắt, chứa địa chỉ của tất cả các thói quen dịch vụ. Bấy (hoặc ngoại lệ) là một ngắt do phần mềm tạo ra do lỗi hoặc yêu cầu của người dùng Hệ điều hành bị gián đoạn.
- Dòng thời gian ngắt

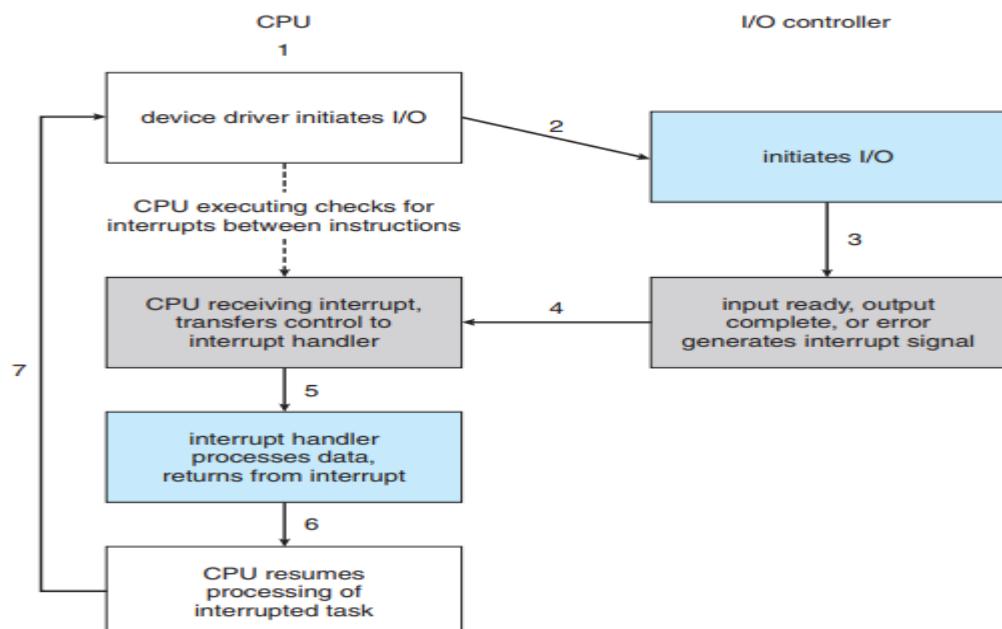


- Xử lý ngắn:

Hệ điều hành duy trì trạng thái của CPU bằng cách lưu trữ các thanh ghi và bộ đếm chương trình (PC). Một số trình điều khiển thiết bị sử dụng ngắn khi tốc độ I / O thấp và chuyển sang bỏ phiếu khi tốc độ tăng đến điểm mà việc bỏ phiếu nhanh hơn và hiệu quả hơn. Xác định loại ngắn đã xảy ra:

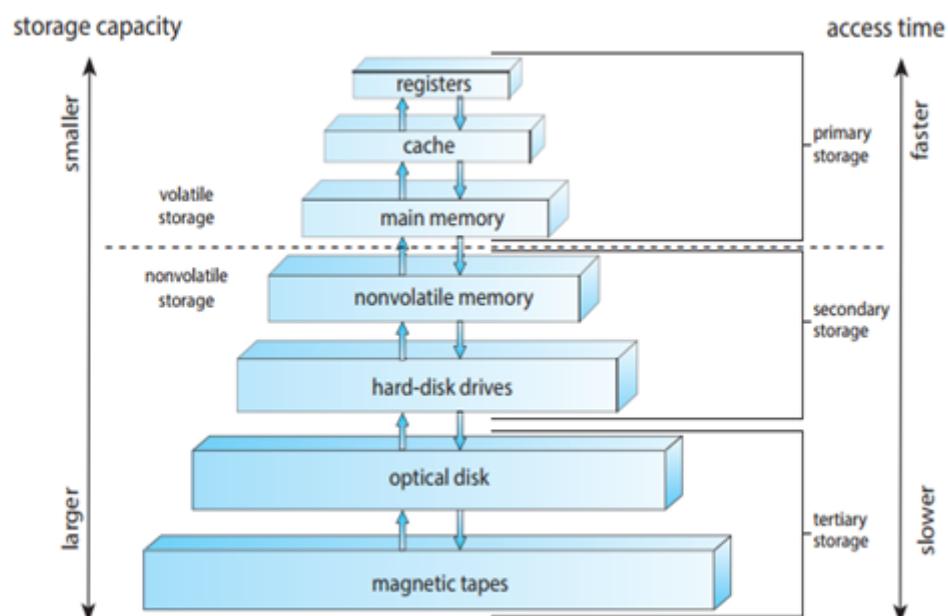
- + Hệ thống ngắn vector được sử dụng để xử lý các sự kiện không đồng bộ và để bẫy các thói quen của chế độ giám sát trong kernel.
- + Các đoạn mã riêng biệt xác định hành động nào nên được thực hiện cho từng loại ngắn.

- Chu kỳ ngắn I/O



## 2.2 Cấu trúc lưu trữ

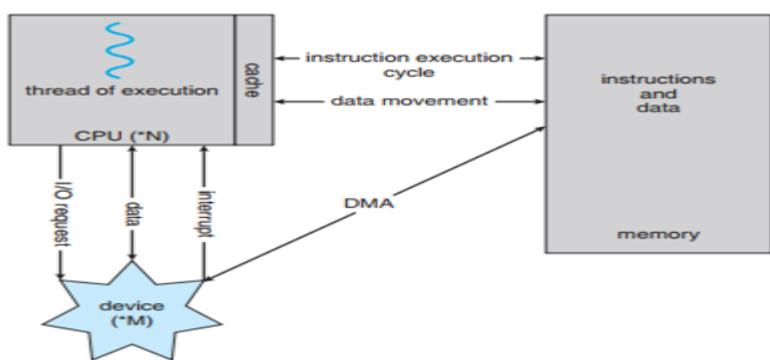
- **Main memory** – only storage media that the CPU can access directly:
  - + Truy cập **ngẫu nhiên**, thường ở dạng Bộ nhớ truy cập **ngẫu nhiên động** (DRAM).
  - + **Dễ bay hơi.**
- Lưu **trữ thứ cấp** - mở rộng bộ nhớ chính cung cấp **dung lượng lưu trữ** lớn không biến đổi.
  - + **Ổ đĩa cứng (HDD)** - đĩa cứng bằng kim loại hoặc thủy tinh được phủ bằng vật liệu ghi từ.
  - + Các thiết bị bộ nhớ **không bay hơi (NVM)** nhanh hơn các ổ đĩa cứng, **không dễ bay hơi.**
- Hệ thống lưu trữ được sắp xếp theo thứ bậc theo: Tốc độ (hoặc thời gian truy cập). Chi phí biến động dung lượng.
- Sự đa dạng của các hệ thống lưu trữ có thể được sắp xếp theo thứ bậc:



## 2.3 Cấu trúc I/O

- Một phần lớn mã hệ điều hành được dành riêng để quản lý I / O, cả hai bởi vì tầm quan trọng của nó đối với độ tin cậy và hiệu suất của một hệ thống và bởi vì tính chất khác nhau của các thiết bị.
- Truy cập bộ nhớ trực tiếp: Được sử dụng cho các thiết bị I / O tốc độ cao có thể truyền thông tin ở gần tốc độ bộ nhớ. Bộ điều khiển thiết bị truyền trực tiếp các khối dữ liệu từ bộ đệm cục bộ sang bộ nhớ chính mà không cần sự can thiệp của CPU. ngắt trên mỗi byte.

Máy tính hiện đại hoạt động như thế nào.



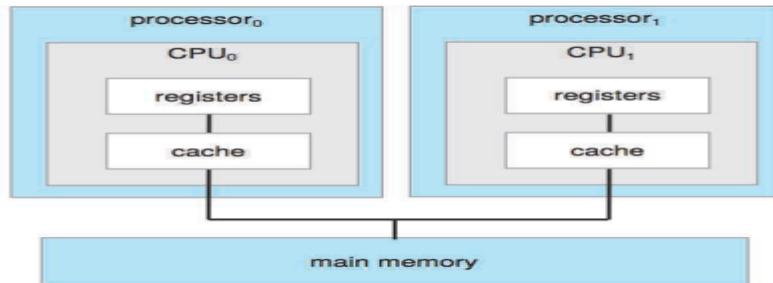
## 3. KIẾN TRÚC HỆ THỐNG MÁY TÍNH

- Một hệ thống máy tính có thể được tổ chức theo một số cách khác nhau, mà chúng ta có thể phân loại đại khái theo số lượng mục đích chung bộ xử lý sử dụng.
- Hệ thống xử lý đơn: Nhiều năm trước, hầu hết các hệ thống máy tính đều sử dụng một bộ xử lý có chứa một CPU với một lõi xử lý duy nhất. Cốt lõi là thành phần thực hiện các hướng dẫn và đăng ký để lưu trữ dữ liệu cục bộ. Một CPU chính với cốt lõi của nó có khả năng thực hiện một tập lệnh đa năng, bao gồm hướng dẫn từ các quy trình. Các hệ thống này có các bộ xử lý mục đích đặc biệt khác là tốt. Chúng có thể ở dạng bộ xử lý dành riêng cho thiết bị, như bộ điều khiển đĩa, bàn phím và đồ họa.

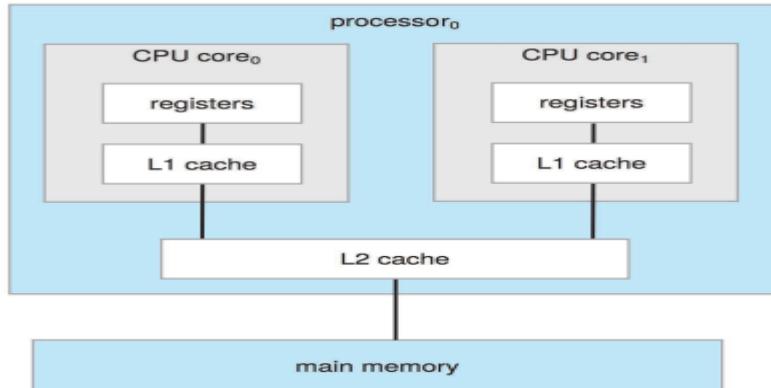
### 3.1 Hệ thống đa xử lý

Trên các máy tính hiện đại, từ thiết bị di động đến máy chủ, các hệ thống đa bộ xử lý hiện đang thống trị toàn cảnh điện toán. Theo truyền thống, các hệ thống như vậy có hai (hoặc nhiều) bộ xử lý, mỗi bộ có CPU một lõi. Các hệ thống đa bộ xử lý phổ biến nhất sử dụng đa xử lý đối xứng (SMP), trong đó

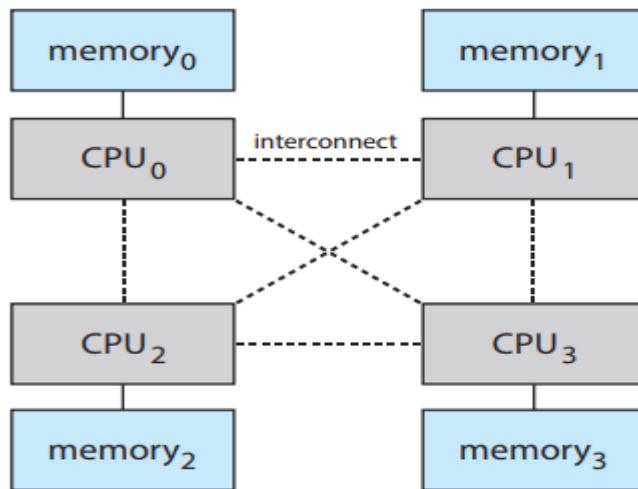
mỗi bộ xử lý CPU ngang hàng thực hiện tất cả các tác vụ, bao gồm chức năng hệ điều hành và quy trình người dùng.



Ngoài ra, một chip có nhiều lõi sử dụng ít năng lượng hơn đáng kể so với nhiều chip đơn lõi, một vấn đề quan trọng đối với thiết bị di động cũng như máy tính xách tay.

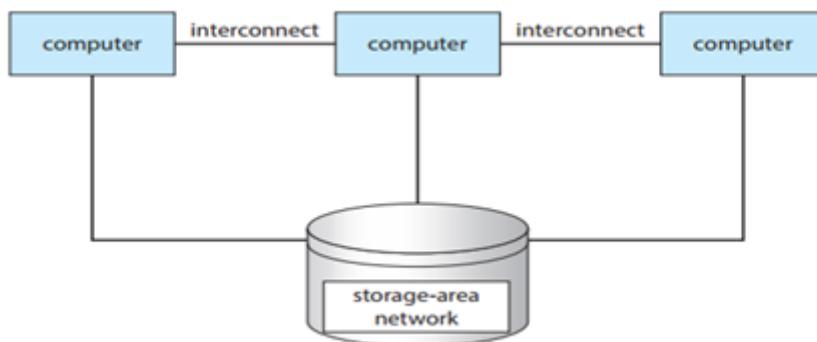


Các CPU được kết nối bằng một kết nối hệ thống chia sẻ, để tất cả các CPU chia sẻ một không gian địa chỉ vật lý. Cách tiếp cận này được gọi là truy cập bộ nhớ không đồng nhất, hoặc NUMA.



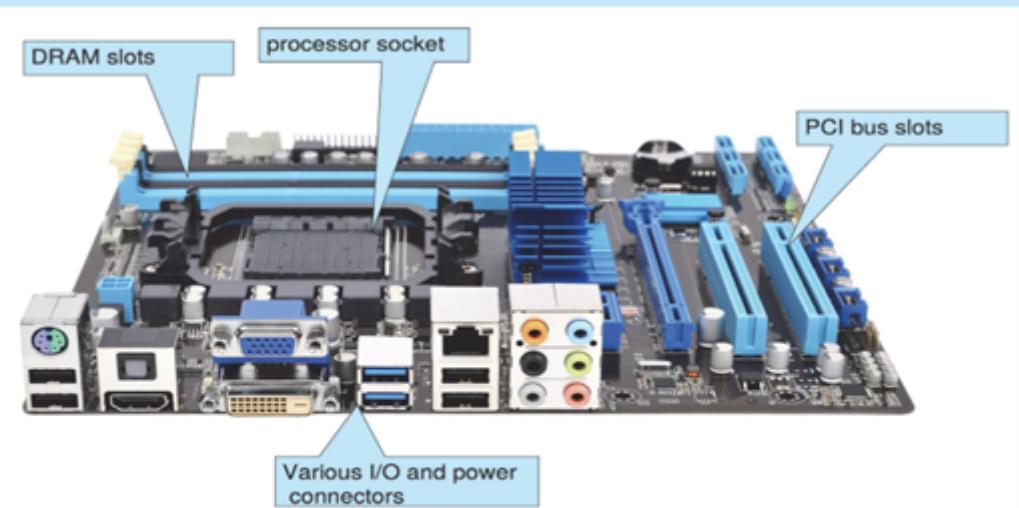
### 3.2 Hệ thống cụm

- Một loại hệ thống đa bộ xử lý khác là một hệ thống phân cụm, tập hợp nhiều CPU. Hệ thống cụm khác với bộ đa xử lý hệ thống. Thông thường chia sẻ lưu trữ thông qua Mạng lưu trữ (SAN) Cung cấp dịch vụ có tính sẵn sàng cao, tồn tại trong các lõi. Phân cụm có thể được cấu trúc không đối xứng hoặc đối xứng. Trong phân cụm không đối xứng, một máy ở chế độ chờ nóng trong khi máy kia đang chạy các ứng dụng. Trong phân cụm đối xứng, hai hoặc nhiều máy chủ đang chạy các ứng dụng và đang theo dõi lẫn nhau. Cấu trúc này rõ ràng là hiệu quả hơn, vì nó sử dụng tất cả các phần cứng có sẵn. Một số cụm được sử dụng cho Máy tính.
- HighPerformance (HPC). Điều này liên quan đến một kỹ thuật được gọi là song song. Một số cụm có Trình quản lý khóa phân tán (DLM) để tránh các hoạt động xung đột.



Bo mạch chủ PC

Consider the desktop PC motherboard with a processor socket shown below:



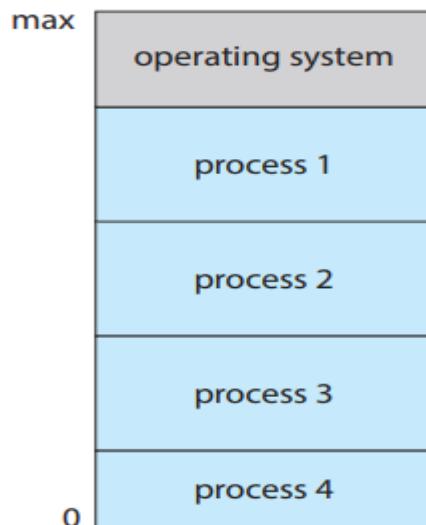
This board is a fully functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general-purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.

#### 4. CÁC THAO TÁC CỦA HỆ ĐIỀU HÀNH

- Một hệ điều hành cung cấp môi trường trong đó các chương trình được thực thi. Bên trong, các hệ điều hành khác nhau rất nhiều, vì chúng được tổ chức theo nhiều dòng khác nhau.
- Ví dụ, để một máy tính bắt đầu chạy chương trình, khi nó được cấp nguồn hoặc được khởi động lại, nó **cần phải có một chương trình ban để chạy**. Như đã lưu ý trước đó, **chương trình ban đầu này, hoặc chương trình bootstrap**, có xu hướng đơn giản. Thường được lưu trữ trong ROM hoặc EPROM, thường được gọi là phần sụn. **Khởi tạo tất cả các khía cạnh của hệ thống Tải kernel hệ điều hành và bắt đầu thực thi**.
- Chương trình **bootstrap** phải biết cách tải hệ điều hành và cách bắt đầu thực thi hệ thống đó. Để thực hiện mục tiêu này, **chương trình bootstrap** phải **xác định vị trí kernel của hệ điều hành và tải nó vào bộ nhớ**.
- Khi kernel được tải và thực thi, nó có thể bắt đầu cung cấp dịch vụ cho hệ thống và người dùng của nó. Một số dịch vụ được cung cấp bên ngoài kernel bởi các chương trình hệ thống được tải vào bộ nhớ khi khởi động để trở thành trình nền hệ thống, chạy toàn bộ thời gian kernel đang chạy.
- Phần cứng bị gián đoạn bởi một trong các thiết bị Ngắt phần mềm (ngoại lệ hoặc bẫy)

#### 4.1 Đa chương trình và đa nhiệm

- Một trong những khía cạnh quan trọng nhất của hệ điều hành là khả năng chạy nhiều chương trình, vì nói chung, một chương trình không thể giữ cho CPU hoặc các thiết bị I / O luôn bận rộn. Hơn nữa, người dùng thường muốn chạy nhiều chương trình cùng một lúc. Đa chương trình làm tăng việc sử dụng CPU, cũng như giữ cho người dùng hài lòng, bằng cách tổ chức các chương trình để CPU luôn có một cái để thực thi. Trong một hệ thống đa chương trình, một chương trình trong thực thi được gọi là một **quá trình**.
- Đa nhiệm là một phần mở rộng hợp lý của đa chương trình. Trong các hệ thống đa nhiệm, CPU thực thi nhiều quy trình bằng cách chuyển đổi giữa chúng, nhưng các công tắc xảy ra thường xuyên, cung cấp cho người dùng thời gian phản hồi nhanh. Hãy xem xét rằng khi một quá trình thực thi, nó thường chỉ thực hiện trong một thời gian ngắn trước khi nó kết thúc hoặc cần thực hiện I / O.
- Bố trí bộ nhớ cho hệ thống đa chương trình

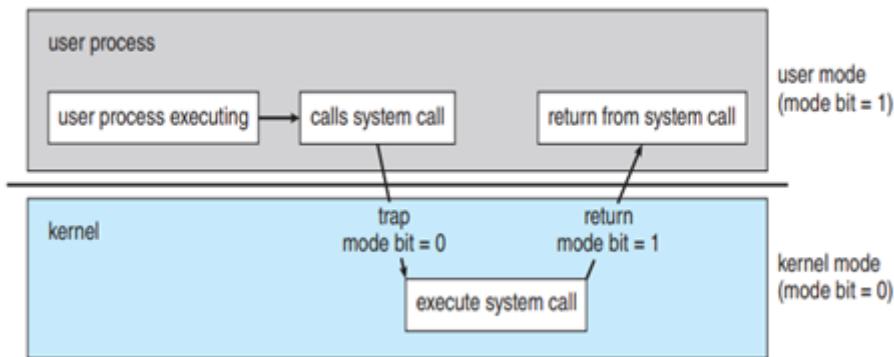


#### 4.2 Hoạt động ở chế độ kép và đa chế độ

- Hoạt động ở chế độ kép cho phép HĐH bảo vệ chính nó và các thành phần hệ thống khác. Ít nhất, chúng ta **cần hai chế độ hoạt động riêng biệt: chế độ người dùng và chế độ nhân** (còn gọi là chế độ giám sát, chế độ hệ thống hoặc chế độ đặc quyền). **Một bit**, được gọi là **bit chế độ**, được thêm vào phần cứng của máy tính để chỉ ra chế độ hiện tại: **kernel (0) hoặc user (1)**. Với bit chế độ, chúng ta

có thể phân biệt giữa một tác vụ được thực hiện thay mặt cho hệ điều hành và một tác vụ được thực hiện thay mặt cho người dùng.

- Chuyển từ chế độ kernel sang chế độ user



### 4.3 Hẹn giờ

Chúng tôi phải đảm bảo rằng hệ điều hành duy trì quyền kiểm soát CPU. Chúng tôi không thể cho phép chương trình người dùng bị kẹt trong một vòng lặp vô hạn hoặc không thể gọi các dịch vụ hệ thống và không bao giờ trả lại quyền điều khiển cho hệ điều hành. Để thực hiện mục tiêu này, chúng ta có thể sử dụng một bộ đếm thời gian. Một bộ đếm thời gian có thể được thiết lập để làm gián đoạn máy tính sau một khoảng thời gian xác định. Khoảng thời gian có thể được cố định (ví dụ: 1/60 giây) hoặc biến (ví dụ: từ 1 mili giây đến 1 giây). Một bộ định thời biến thường được thực hiện bởi đồng hồ tốc độ cố định và bộ đếm. Hệ điều hành đặt bộ đếm. Mỗi khi đồng hồ tích tắc, bộ đếm bị giảm. Khi bộ đếm đạt 0, một ngắt xảy ra. Chẳng hạn, bộ đếm 10 bit với đồng hồ 1 mili giây cho phép ngắt theo các khoảng từ 1 mili giây đến 1.024 mili giây, trong các bước 1 mili giây.

## 5. QUẢN LÝ TÀI NGUYÊN

Như chúng ta đã thấy, một hệ điều hành là một trình quản lý tài nguyên. CPU CPU hệ thống, không gian bộ nhớ, không gian lưu trữ tệp và thiết bị I / O là một trong những tài nguyên mà hệ điều hành phải quản lý.

### 5.1 Quản lý quy trình

Một quá trình là một chương trình trong thực thi. Nó là một đơn vị công việc trong hệ thống. Chương trình là một thực thể thụ động, quá trình là một thực thể hoạt động. Quá trình cần tài nguyên để thực hiện nhiệm vụ của nó (CPU, bộ

nhớ, I / O, tệp, dữ liệu khởi tạo, Máy) Quy trình đơn luồng có một bộ đếm chương trình xác định vị trí của lệnh tiếp theo để thực thi. Quá trình đa luồng có một bộ đếm chương trình trên mỗi luồng Thông thường hệ thống có nhiều quy trình (một số người dùng và một số quy trình hệ điều hành) chạy đồng thời trên một hoặc nhiều CPU Đồng thời bằng cách ghép các CPU giữa các quy trình / luồng Hệ điều hành chịu trách nhiệm cho các hoạt động sau liên quan đến quản lý quy trình:

- Tạo và xóa cả quá trình người dùng và hệ thống.
- Định chỉ và tiếp tục quá trình.
- Cung cấp các cơ chế để đồng bộ hóa quá trình.
- Cung cấp cơ chế truyền thông quá trình.
- Cung cấp cơ chế xử lý bế tắc.

### 5.2 Quản lý bộ nhớ

Để thực thi một chương trình, Tất cả (hoặc một phần) các hướng dẫn phải nằm trong bộ nhớ Tất cả (hoặc một phần) dữ liệu cần thiết của chương trình phải nằm trong bộ nhớ. Quản lý bộ nhớ xác định những gì trong bộ nhớ và khi Tối ưu hóa việc sử dụng CPU và phản ứng của máy tính với người dùng Hệ điều hành chịu trách nhiệm cho các hoạt động sau liên quan đến quản lý bộ nhớ:

- Theo dõi phần nào của bộ nhớ hiện đang được sử dụng và quá trình nào đang sử dụng chúng.
- Phân bổ và giải phóng không gian bộ nhớ khi cần thiết.
- Quyết định quy trình nào (hoặc một phần của quy trình) và dữ liệu sẽ chuyển vào và hết bộ nhớ

### 5.3 Quản lý hệ thống tệp

Quản lý tập tin là một trong những thành phần dễ thấy nhất của một hệ điều hành. Máy tính có thể lưu trữ thông tin trên một số loại phương tiện vật lý khác nhau. Lưu trữ thứ cấp là lưu trữ phổ biến nhất, nhưng cũng có thể lưu trữ cấp ba: Các tệp thường được tổ chức vào các thư mục Kiểm soát truy cập trên hầu hết các hệ thống để xác định ai có thể truy cập những gì. Hệ điều hành chịu trách nhiệm cho các hoạt động sau liên quan đến quản lý tệp:

- Tạo và xóa tập tin.
- Tạo và xóa các thư mục để tổ chức các tập tin.

- Hỗ trợ nguyên thủy để thao tác các tập tin và thư mục.
- Ánh xạ tập tin vào bộ lưu trữ lớn.
- Sao lưu tệp trên phương tiện lưu trữ ổn định (không biến đổi).

#### 5.4 Quản lý lưu trữ lớn

Thông thường các đĩa được sử dụng để lưu trữ các chương trình và dữ liệu không phù hợp với bộ nhớ chính hoặc phải được lưu giữ trong một khoảng thời gian của Long lâu Quản lý đúng cách có tầm quan trọng trung tâm Toàn bộ tốc độ hoạt động của máy tính bần lề trên hệ thống con đĩa và thuật toán của nó. Hệ điều hành chịu trách nhiệm sau đây các hoạt động liên quan đến quản lý lưu trữ thứ cấp:

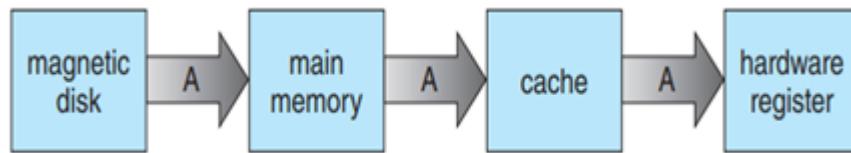
- Gắn kết và ngắt kết nối
- Quản lý không gian trống
- Phân bổ lưu trữ
- Lập lịch đĩa
- Phân vùng
- Bảo vệ

#### 5.5 Quản lý cache

Bộ nhớ đệm là một nguyên tắc quan trọng của hệ thống máy tính. Ở đây, cách thức hoạt động của nó. Thông tin thường được lưu trong một số hệ thống lưu trữ (như bộ nhớ chính). Khi được sử dụng, nó được sao chép vào một hệ thống lưu trữ nhanh hơn trong bộ nhớ cache tạm thời. Bộ nhớ cache nhỏ hơn bộ nhớ được lưu trong bộ nhớ cache Quản lý bộ đệm là một vấn đề thiết kế quan trọng Kích thước bộ đệm và chính sách thay thế  
Có nhiều cách lưu trữ khác nhau

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Môi trường đa nhiệm phải cẩn thận để sử dụng giá trị gần đây nhất, bất kể nó được lưu trữ ở đâu trong hệ thống phân cấp lưu trữ. Môi trường đa xử lý phải cung cấp sự kết hợp bộ đệm trong phần cứng sao cho tất cả các CPU có giá trị gần đây nhất trong bộ đệm của chúng trong môi trường phân tán, tình huống thậm chí còn nhiều hơn phức tạp: Một số bản sao của dữ liệu có thể tồn tại nhiều giải pháp khác nhau.



## 5.6 Quản lý hệ thống I/O

Một trong những mục đích của một hệ điều hành là để che giấu những đặc thù của thiết bị phần cứng cụ thể từ người dùng. Ví dụ, trong UNIX, các đặc thù của thiết bị I/O bị ẩn khỏi phần lớn hệ điều hành bởi chính hệ thống con I/O. Hệ thống con I/O bao gồm một số thành phần:

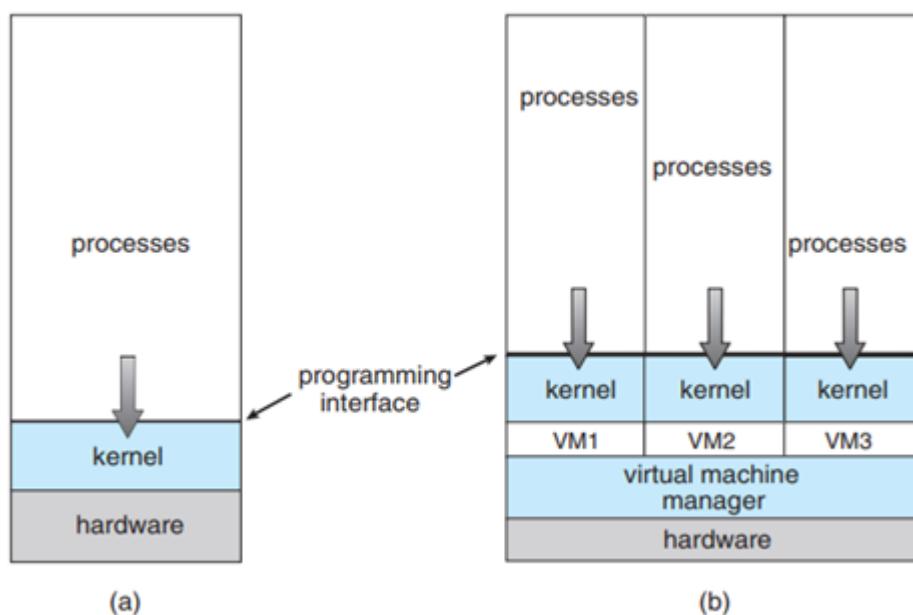
- Thành phần quản lý bộ nhớ bao gồm bộ đệm, bộ đệm và spooling.
- Giao diện trình điều khiển thiết bị chung.
- Trình điều khiển cho các thiết bị phần cứng cụ thể

## 6. TÍNH AN TOÀN VÀ BẢO MẬT

- Bảo vệ, là bất kỳ cơ chế nào để kiểm soát truy cập của các quy trình hoặc người dùng đến các tài nguyên được xác định bởi một hệ thống máy tính. Cơ chế này phải cung cấp các phương tiện để chỉ định các điều khiển được áp đặt và để thực thi các điều khiển. Bảo mật - bảo vệ một hệ thống khỏi các cuộc tấn công bên ngoài và bên trong.
- Các hệ thống thường phân biệt đầu tiên giữa những người dùng, để xác định ai có thể làm gì.
- Danh tính người dùng (UID hoặc ID bảo mật) bao gồm tên và số liên quan. ID người dùng sau đó được liên kết với tất cả các tệp, quy trình của người dùng đó để xác định kiểm soát truy cập.
- Mã định danh nhóm (GID) cho phép xác định nhóm người dùng để kiểm soát truy cập, sau đó cũng được liên kết với từng quy trình hoặc tệp.
- Leo thang đặc quyền cho phép người dùng thay đổi thành ID hiệu quả với nhiều quyền hơn.

## 7. ẢO HÓA

- Ảo hóa là một công nghệ cho phép chúng ta trùu tượng hóa phần cứng của một máy tính (CPU, bộ nhớ, ổ đĩa, thẻ giao diện mạng, v.v.) vào một số môi trường thực thi khác nhau, từ đó tạo ra ảo giác rằng mỗi môi trường riêng biệt đang chạy máy tính riêng của nó.
- Các môi trường này có thể được xem là các hệ điều hành riêng lẻ khác nhau (ví dụ: Windows và UNIX) có thể đang chạy cùng lúc và có thể tương tác với nhau. Người dùng máy ảo có thể chuyển đổi giữa các hệ điều hành khác nhau giống như cách người dùng có thể chuyển đổi giữa các quy trình khác nhau chạy đồng thời trong một hệ điều hành.
- Mô phỏng được sử dụng khi loại CPU nguồn khác với loại CPU đích: Phương pháp chậm nhất nói chung Khi ngôn ngữ máy tính không được biên dịch thành mã gốc.
- Các trường hợp sử dụng liên quan đến máy tính xách tay và máy tính để bàn chạy nhiều hệ điều hành để khám phá hoặc tương thích. VMM có thể chạy tự nhiên, trong trường hợp đó chúng cũng là máy chủ.



## 8. HỆ THỐNG PHÂN TÁN

- Một hệ thống phân tán là một tập hợp các hệ thống máy tính không đồng nhất về mặt vật lý, có thể không đồng nhất được nối mạng để cung cấp cho người dùng quyền truy cập vào các tài nguyên khác nhau mà hệ thống duy trì. Truy cập vào tài nguyên được chia sẻ làm tăng tốc độ tính toán, chức năng, tính khả

**dụng của dữ liệu và độ tin cậy.** Một số hệ điều hành tổng quát truy cập mạng dưới dạng truy cập tệp, với các chi tiết về mạng có trong trình điều khiển thiết bị giao diện mạng. Network is communications paths (TCP/IP is most common protocol stack

- Local Area Network (LAN)
  - Wide Area Network (WAN)
  - Metropolitan Area Network (MAN)
  - Personal Area Network (PAN)
- Hệ điều hành mạng là hệ điều hành cung cấp các tính năng chẵng hạn như chia sẻ tệp trên mạng, cùng với sơ đồ truyền thông cho phép các quá trình khác nhau trên các máy tính khác nhau để trao đổi tin nhắn. Một máy tính chạy hệ điều hành mạng hoạt động tự chủ từ tất cả các máy tính khác trên mạng, mặc dù biết về mạng và có khả năng giao tiếp với các máy tính nối mạng khác.

## 9. CẤU TRÚC DỮ LIỆU KERNEL

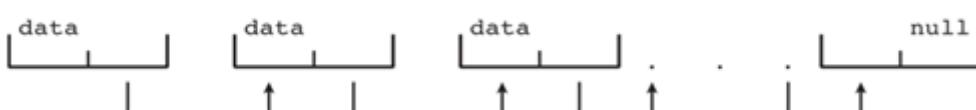
Chúng tôi chuyển sang một chủ đề trung tâm để thực hiện hệ điều hành: cách dữ liệu được cấu trúc trong hệ thống. Trong phần này, chúng tôi mô tả ngắn gọn một số cấu trúc dữ liệu cơ bản được sử dụng rộng rãi trong các hệ điều hành.

### 9.1 Lists, Stacks, and Queue

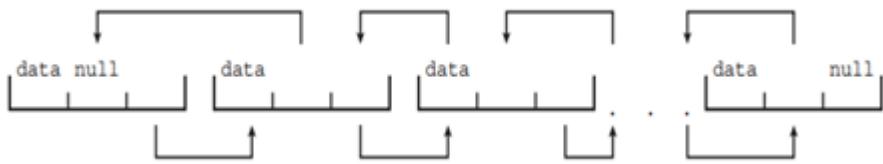
Sau mảng, danh sách có lẽ là cấu trúc dữ liệu cơ bản nhất trong khoa học máy tính. Trong khi mỗi mục trong một mảng có thể được truy cập trực tiếp, các mục đó trong một danh sách phải được truy cập theo một thứ tự cụ thể. Đó là, một danh sách đại diện cho một tập hợp các giá trị dữ liệu dưới dạng một chuỗi. Phương pháp phổ biến nhất để thực hiện cấu trúc này là một danh sách liên kết, trong đó các mục được liên kết với nhau. Liên kết danh sách có nhiều loại:

- Trong một danh sách liên kết đơn, mỗi mục chỉ đến người kế nhiệm của nó.

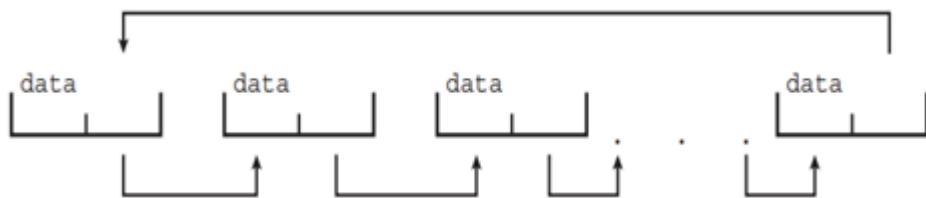
#### ***Singly linked list***



- Trong danh sách được liên kết đôi, một mục nhất định có thể tham chiếu đến người tiền nhiệm của nó hoặc người kế vị của nó.



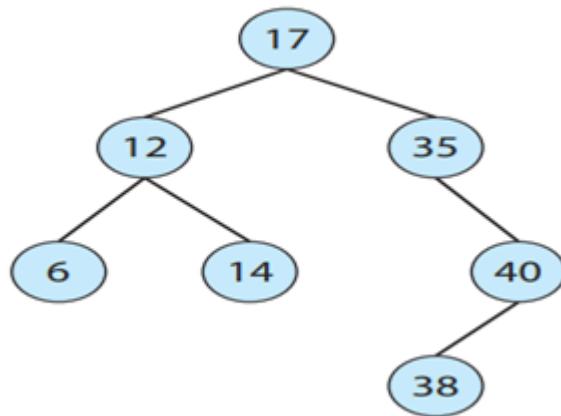
- Trong danh sách được liên kết vòng tròn, phần tử cuối cùng trong danh sách đề cập đến phần đầu tiên phần tử, thay vì null.



- Một ngăn xếp là một cấu trúc dữ liệu được sắp xếp theo thứ tự sử dụng cái cuối cùng, đầu tiên. Nguyên tắc ra (LIFO) để thêm và xóa các mục, nghĩa là mục cuối cùng được đặt trên ngăn xếp là mục đầu tiên bị xóa. Ngược lại, một hàng đợi là một cấu trúc dữ liệu được sắp xếp theo thứ tự sử dụng Nguyên tắc nhập trước, xuất trước (FIFO): các mục được xóa khỏi hàng đợi theo thứ tự trong đó chúng được chèn vào.

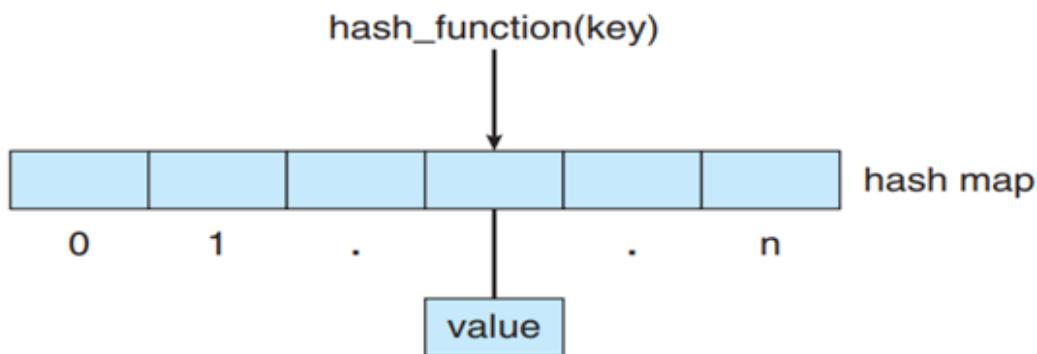
## 9.2 Trees

Cây là một cấu trúc dữ liệu có thể được sử dụng để biểu diễn dữ liệu theo cấp bậc. Dữ liệu các giá trị trong cấu trúc cây được liên kết thông qua các mối quan hệ con cha mẹ. Trong một Cây nói chung, cha mẹ có thể có số lượng con không giới hạn. Trong một nhị phân cây, cha mẹ có thể có nhiều nhất hai con, mà chúng tôi gọi là con trái và đúng đứa trẻ. Một cây tìm kiếm nhị phân cũng yêu cầu một thứ tự giữa cha mẹ. Lỗi có hai con trong đó con trái  $\leq$  con phải.



### 9.3 Hash Functions and Maps

Hàm băm lấy dữ liệu làm đầu vào của nó, thực hiện thao tác số trên dữ liệu và trả về một giá trị số. Một cách sử dụng hàm băm là để thực hiện một bản đồ băm, liên kết (hoặc ánh xạ) các cặp [key: value] bằng cách sử dụng hàm băm



9.4

### Bitmaps

Một bitmap là một chuỗi gồm n chữ số nhị phân có thể được sử dụng để thể hiện trạng thái của n mặt hàng.

## 10. CÁC MÔI TRƯỜNG MÁY TÍNH

Cho đến nay, chúng tôi đã mô tả ngắn gọn một số khía cạnh của hệ thống máy tính và hệ điều hành quản lý chúng. Bây giờ chúng ta chuyển sang một cuộc thảo luận về cách hệ điều hành được sử dụng trong nhiều môi trường máy tính.

### 10.1 Máy tính truyền thống

- Máy đa năng độc lập.

- Máy tính di động kết nối qua mạng không dây.
- Cổng cung cấp quyền truy cập web

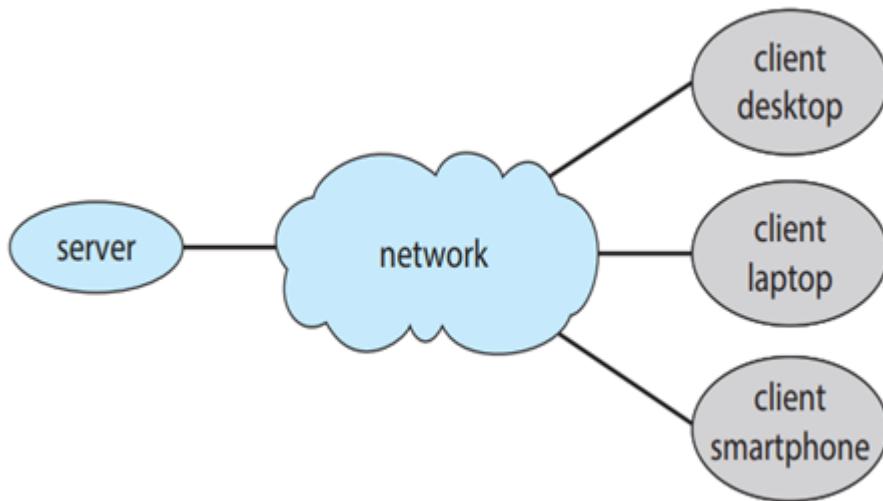
## 10.2 Mô hình Di động

Điện toán di động đề cập đến điện toán trên điện thoại thông minh cầm tay và máy tính bảng máy tính. Các thiết bị này chia sẻ các đặc điểm vật lý khác biệt của xách tay và nhẹ. Chẳng hạn như điện thoại thông minh cầm tay, máy tính bảng, v.v. Hai hệ điều hành hiện đang thống trị điện toán di động: Apple iOS và Google Android.

## 10.3 Mô hình client-server

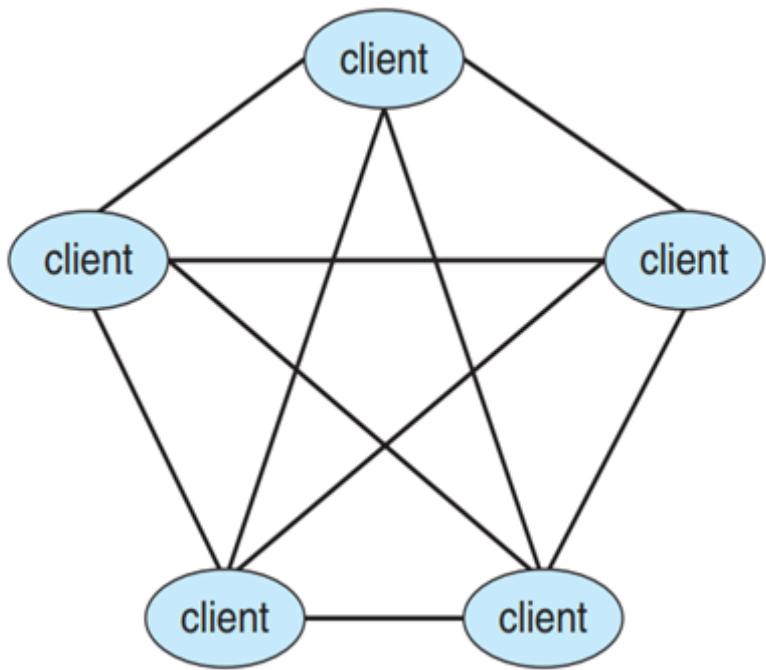
Kiến trúc mạng hiện đại có các sắp xếp trong đó máy chủ hệ thống đáp ứng yêu cầu được tạo bởi hệ thống máy khách. Hình thức chuyên ngành này hệ thống phân tán, được gọi là hệ thống máy chủ client, có cấu trúc chung.

Hệ thống máy chủ có thể được phân loại thành máy chủ và tệp tính toán máy chủ: Hệ thống máy chủ tính toán cung cấp giao diện cho khách hàng để yêu cầu dịch vụ (nghĩa là cơ sở dữ liệu) Hệ thống máy chủ tệp cung cấp giao diện cho khách hàng lưu trữ và truy xuất tệp



## 10.4 Mô hình Peer-to-Peer

Một mô hình khác của hệ thống phân tán P2P không phân biệt máy khách và máy chủ Ví dụ bao gồm Napster và Gnutella, Thoại qua IP (VoIP) như Skype

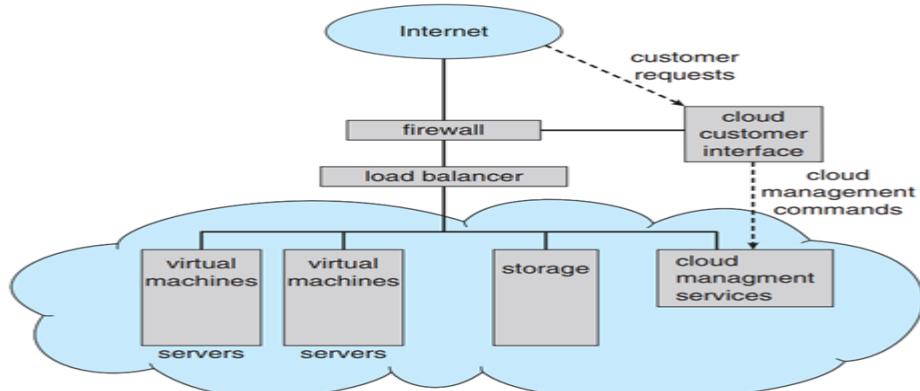


## 10.5 Mô hình đám mây

Điện toán đám mây là một loại điện toán cung cấp tính toán, lưu trữ, và thậm chí các ứng dụng như một dịch vụ trên toàn mạng. Theo một số cách, nó LỚN mở rộng logic của ảo hóa, bởi vì nó sử dụng ảo hóa làm cơ sở cho chức năng của nó. Ví dụ: Đám mây điện toán đám mây Amazon (ec2). Có nhiều loại dịch vụ:

- Phần mềm dưới dạng dịch vụ (SaaS) - một hoặc nhiều ứng dụng có sẵn qua Internet
- Nền tảng Internet dưới dạng Dịch vụ(PaaS) - ngăn xếp phần mềm đã sẵn sàng để sử dụng ứng dụng thông qua.

- Cơ sở hạ tầng Internet dưới dạng dịch vụ (IaaS) - máy chủ hoặc bộ lưu trữ có sẵn qua Internet (tức là, bộ nhớ có sẵn để sử dụng sao lưu) - Nhiều loại cấu trúc.



## 10.6 Hệ thống nhúng thời gian thực

Máy tính nhúng là hình thức phổ biến nhất của máy tính tồn tại. Những thiết bị này được tìm thấy ở khắp mọi nơi, từ động cơ xe hơi và sản xuất robot vào ổ đĩa quang và lò vi sóng. Họ có xu hướng rất cụ thể nhiệm vụ. Các hệ thống họ chạy trên thường là nguyên thủy, và do đó, hoạt động hệ thống cung cấp các tính năng hạn chế. Thông thường, họ có ít hoặc không có giao diện người dùng, thích dành thời gian của họ để theo dõi và quản lý các thiết bị phần cứng, chẳng hạn như động cơ ô tô và cánh tay robot. Các hệ thống nhúng hầu như luôn chạy các hệ điều hành thời gian thực. Một hệ thống thời gian thực có các ràng buộc thời gian cố định được xác định rõ.

## 11. HOẠT ĐỘNG MIỄN PHÍ VÀ NGUỒN MỞ

Việc nghiên cứu các hệ điều hành đã được thực hiện dễ dàng hơn nhờ có sẵn một số lượng lớn các phần mềm miễn phí và các bản phát hành nguồn mở. Cả hệ điều [http://gnu.org/philatics/open-source-misses-the-point.html/](http://gnu.org/philatics/open-source-misses-the-point.html) cho một thảo luận về chủ đề này) Chống lại phong trào bảo vệ bản sao và Quản lý quyền kỹ thuật số (DRM) được bắt đầu bởi Tổ chức phần mềm tự do (FSF), có Giấy phép công cộng GNU copyleft GNU GNU (GPL) hoặc GPL (LGPL)

## 12. TỔNG KẾT

- Hệ điều hành là phần mềm quản lý phần cứng máy tính, cũng như cung cấp môi trường cho các chương trình ứng dụng chạy.
- Ngắt là một cách quan trọng trong đó phần cứng tương tác với hệ điều hành. Một thiết bị phần cứng kích hoạt ngắt bằng cách gửi tín hiệu đến CPU để cảnh báo CPU rằng một số sự kiện cần chú ý. Ngắt được quản lý bởi trình xử lý ngắt.
- Lưu trữ không biến đổi là một phần mở rộng của bộ nhớ chính và có khả năng chứa một lượng lớn dữ liệu vĩnh viễn.
- Thiết bị lưu trữ không biến đổi phổ biến nhất là đĩa cứng, có thể cung cấp lưu trữ cả chương trình và dữ liệu.
- Sự đa dạng của các hệ thống lưu trữ trong một hệ thống máy tính có thể được sắp xếp theo thứ bậc theo tốc độ và chi phí. Các mức cao hơn là đắt tiền, nhưng chúng là nhanh chóng. Khi chúng ta chuyển xuống hệ thống phân cấp, chi phí cho mỗi bit thường giảm, trong khi thời gian truy cập thường tăng.
- Kiến trúc máy tính hiện đại là các hệ thống đa bộ xử lý, trong đó mỗi CPU chứa một số lõi máy tính.

- Để sử dụng CPU tốt nhất, các hệ điều hành hiện đại sử dụng đa chương trình, cho phép một số công việc nằm trong bộ nhớ cùng một lúc, do đó đảm bảo CPU luôn có công việc để thực thi.
- Đa nhiệm là một phần mở rộng của đa chương trình trong đó các thuật toán lập lịch CPU chuyển đổi nhanh chóng giữa các quy trình, cung cấp cho người dùng thời gian phản hồi nhanh.
- Để ngăn các chương trình người dùng can thiệp vào hoạt động đúng đắn của hệ thống, phần cứng hệ thống có hai chế độ: chế độ người dùng và chế độ kernel.
- Các hướng dẫn khác nhau được đặc quyền và chỉ có thể được thực thi trong chế độ kernel. Các ví dụ bao gồm hướng dẫn để chuyển sang chế độ kernel, điều khiển I / O, quản lý hẹn giờ và quản lý ngắt.
- Một quy trình là đơn vị công việc cơ bản trong một hệ điều hành. Quản lý quy trình bao gồm tạo và xóa các quy trình và cung cấp các cơ chế để các quy trình giao tiếp và đồng bộ hóa với nhau.
- Một hệ điều hành quản lý bộ nhớ bằng cách theo dõi những phần nào của bộ nhớ đang được sử dụng và của ai. Nó cũng chịu trách nhiệm phân bổ động và giải phóng không gian bộ nhớ.
- Không gian lưu trữ được quản lý bởi hệ điều hành; điều này bao gồm việc cung cấp các hệ thống tệp để thể hiện các tệp và thư mục và quản lý không gian trên các thiết bị lưu trữ lớn.
- Hệ điều hành cung cấp các cơ chế bảo vệ và bảo mật hệ điều hành và người dùng. Các biện pháp bảo vệ kiểm soát quyền truy cập của các quy trình hoặc người dùng vào các tài nguyên có sẵn của hệ thống máy tính.
- Điện toán diễn ra trong nhiều môi trường khác nhau, bao gồm điện toán truyền thống, điện toán di động, hệ thống máy chủ-máy khách, hệ thống ngang hàng, điện toán đám mây và hệ thống nhúng thời gian thực.
- Các hệ điều hành miễn phí và nguồn mở có sẵn ở định dạng sourcecode. Phần mềm miễn phí được cấp phép để cho phép sử dụng miễn phí, phân phối lại và sửa đổi. GNU / Linux, FreeBSD và Solaris là những ví dụ về các hệ thống nguồn mở phổ biến.

### 13. QUESTIONS

1. Thành phần nào sau đây là 1 bộ phận của hệ điều hành ?

A.Cell

B.FAT

C.Kernel

D.Tất cả đáp án trên đều đúng

2. Lớp nào sau đây là lớp phần mềm gần với lớp phần cứng nhất?

A.Hệ điều hành

B.Phần mềm ứng dụng

C.Trình biên dịch

D.Loader

3. Hệ điều hành chịu trách nhiệm ?

A.Cung cấp tài nguyên theo yêu cầu của người dùng hoặc ứng dụng

B.Cung cấp các thiết bị thiết lập ban đầu

C.Dịch vụ được yêu cầu cho việc liên lạc qua mạng

D. Tất cả những điều trên

4. Khi chúng ta tắt máy tính, dữ liệu nào sau đây bị mất ?

A.Non-Volatile memory

B.Volatile memory

C.Tất cả đáp án trên đều đúng

D.Tất cả đáp án trên đều sai

5. Dịch vụ điện toán đám mây nào sau đây chiếm tỉ trọng cao nhất trên thị trường ?

A.AWS

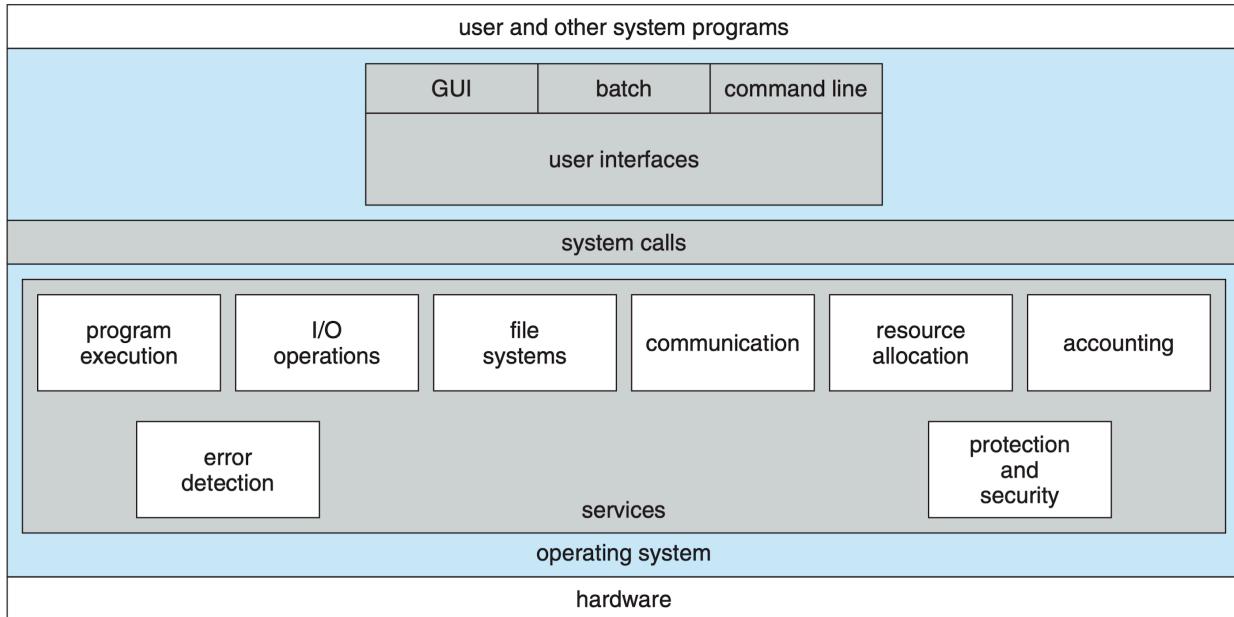
. Azure

C.GCP

D.Tất cả đáp án trên đều sai



# CHƯƠNG 2: CẤU TRÚC CỦA HỆ ĐIỀU HÀNH



## 1. DỊCH VỤ CỦA HỆ ĐIỀU HÀNH

Hệ điều hành cung cấp các dịch vụ:

- Giao diện người dùng (User Interface)**: Cung cấp phương tiện để giao tiếp với người dùng. Tùy thuộc vào hệ thống, đây có thể là giao diện command line (ví dụ: sh, csh, ksh, tcsh, v.v.), giao diện GUI (ví dụ: Windows, X-Windows, KDE, Gnome, v.v.) hoặc hệ thống lệnh batch
- Thực thi chương trình (Program Execution)**: HĐH phải có khả năng tải chương trình vào RAM, chạy chương trình và chấm dứt chương trình, bình thường hoặc bất thường.
- Hoạt động I/O (I/O Operations)**: HĐH chịu trách nhiệm truyền dữ liệu đến và từ các thiết bị I/O, bao gồm bàn phím, thiết bị đầu cuối, máy in và thiết bị lưu trữ.
- Thao tác hệ thống tệp (File-System Manipulation)**: Ngoài lưu trữ dữ liệu thô, HĐH còn chịu trách nhiệm duy trì cấu trúc thư mục và thư mục con, ánh xạ tên tệp vào các khối lưu trữ dữ liệu cụ thể và cung cấp các công cụ để điều hướng và sử dụng hệ thống tệp.

5. **Giao tiếp (Communications):** giao tiếp giữa các quá trình, IPC, giữa các tiến trình chạy trên cùng một bộ xử lý hoặc giữa các tiến trình chạy trên các bộ xử lý riêng biệt hoặc các máy riêng biệt. Có thể được triển khai dưới dạng bộ nhớ chia sẻ hoặc gửi thông điệp, (hoặc một số hệ thống có thể cung cấp cả hai.)
6. **Phát hiện lỗi (Error Detection):** Cả hai lỗi phần cứng và phần mềm phải được phát hiện và xử lý phù hợp, tối thiểu các hậu quả có hại. Một số hệ thống có thể bao gồm các hệ thống phòng tránh hoặc khôi phục lỗi phức tạp, bao gồm sao lưu, ổ RAID và các hệ thống dự phòng khác. Công cụ gỡ lỗi và chẩn đoán hỗ trợ người dùng và quản trị viên truy tìm nguyên nhân của sự cố.

Một số chức năng hệ điều hành có mặt để đảm bảo hoạt động hiệu quả của chính nó:

- **Phân bổ tài nguyên (Resource Allocation):** Ví dụ: Chu kỳ CPU, bộ nhớ chính, không gian lưu trữ và các thiết bị ngoại vi. Một số tài nguyên được quản lý với các hệ thống chung và các tài nguyên khác, được thiết kế rất cẩn thận và được điều chỉnh đặc biệt cho một tài nguyên và môi trường hoạt động cụ thể
- **Kiểm toán (Accounting):** Theo dõi hoạt động của hệ thống và sử dụng tài nguyên, cho mục đích sử dụng hoặc lưu giữ hồ sơ thống kê có thể được dùng để tối ưu hóa hiệu suất trong tương lai.
- **Bảo vệ và Bảo mật (Protection and Security):** Ngăn chặn tác hại đối với hệ thống và tài nguyên, đến từ các process nội bộ hoặc tác nhân bên ngoài độc hại. Xác thực, quyền sở hữu và quyền truy cập hạn chế là những phần rõ ràng của hệ thống này. Các hệ thống bảo mật cao có thể ghi lại tất cả các hoạt động của quy trình một cách chi tiết rõ ràng và quy định bảo mật.

## 2. GIAO DIỆN NGƯỜI DÙNG CỦA HỆ ĐIỀU HÀNH

### 2.1 Command Interpreter (CI)

- Chức năng chính là nhận, xử lý yêu cầu của người dùng và khởi chạy các chương trình được yêu cầu.
- Trong một số hệ thống, CI có thể được tích hợp trực tiếp vào kernel.

- Thông thường, CI là một chương trình riêng biệt và khởi chạy một khi người dùng đăng nhập hoặc truy cập hệ thống.

Ví dụ như UNIX cung cấp cho người dùng lựa chọn các shell khác nhau (Bourne shell, C shell, Bourne-Again shell, Korn shell,...) có thể được cấu hình để tự động khởi chạy khi đăng nhập hoặc có thể được thay đổi nhanh chóng.

- Các shell khác nhau cung cấp chức năng khác nhau theo các lệnh nhất định được shell trực tiếp triển khai mà không khởi chạy bất kỳ chương trình bên ngoài nào.

## 2.2 Graphical User Interfaces (GUI)

- Thường được triển khai như một phép ẩn dụ trên desktop, với các folder, thùng rác và biểu tượng tài nguyên.
- Các icon đại diện cho một số mục trên hệ thống và phản hồi tương ứng khi icon được kích hoạt.
- Được phát triển lần đầu tiên vào đầu những năm 1970 tại cơ sở nghiên cứu của Xerox PARC.
- Trong một số hệ thống, GUI chỉ là giao diện người dùng để kích hoạt trình thông dịch dòng lệnh truyền thống đang chạy trong nền. Nói cách khác, GUI là một vỏ đồ họa thực sự theo đúng nghĩa của nó.
- Mac thường chỉ cung cấp giao diện GUI. Tuy nhiên, với sự ra đời của OSX (dựa một phần vào UNIX), Command Interpreter cũng đã có mặt.
- Bởi vì chuột và bàn phím không thực tế đối với các thiết bị di động nhỏ, ngày nay chúng thường sử dụng giao diện màn hình cảm ứng, đáp ứng các kiểu vuốt hoặc "cử chỉ" khác nhau. Lúc ban đầu, các thiết bị này có bàn phím vật lý, tuy nhiên hiện tại đa số đã chuyển sang bàn phím cảm ứng trong màn hình.

## 2.3 Choice of interface

- Hầu hết các hệ thống hiện đại cho phép người dùng cá nhân chọn giao diện mong muốn và tùy chỉnh hoạt động của nó, cũng như khả năng chuyển đổi giữa các giao diện khác nhau khi cần.

- Giao diện GUI thường cung cấp tùy chọn cho cửa sổ giả lập terminal để nhập lệnh dòng lệnh.
- Các lệnh dòng lệnh cũng có thể được nhập vào các kịch bản shell scripts, sau đó có thể được chạy như bất kỳ chương trình nào khác.

### 3. LỜI GỌI HỆ THỐNG (SYSTEM CALL)

- System calls cung cấp phương tiện để người dùng hoặc các chương trình ứng dụng gọi các dịch vụ của hệ điều hành.
- Thường được viết bằng C hoặc C++, một số được viết bằng assembly để có hiệu suất tối ưu.
- Ta có thể sử dụng "strace" để xem thêm các ví dụ về số lượng lớn các system call được gọi bằng một lệnh đơn giản. VD:

```
$ cc hello.c
```

```
$ strace ./a.out
```

- Hầu hết các lập trình viên không sử dụng trực tiếp system calls cấp thấp mà thay vào đó sử dụng API (Giao diện lập trình ứng dụng).

VD: Khi người dùng gọi hàm CreateProcess() trong Window, thực chất là system call NTCREATEPROCESS() sẽ được gọi

- Việc sử dụng API thay vì trực tiếp system call cung cấp khả năng di động hơn giữa các hệ thống khác nhau. API thực hiện các system call phù hợp thông qua giao diện hệ thống, sử dụng tra cứu bảng để truy cập các cuộc gọi hệ thống được đánh số cụ thể.
- Cách truyền tham số cho hệ điều hành đơn giản nhất là truyền các tham số trong thanh ghi. Tuy nhiên, trong một số trường hợp, có thể có nhiều tham số hơn các thanh ghi. Trong các trường hợp này, các tham số thường được lưu trữ trong một khối hoặc bảng, trong bộ nhớ và địa chỉ của khối được truyền dưới dạng tham số trong thanh ghi

## 4. PHÂN LOẠI SYSTEM CALL

Có thể phân loại system call vào 6 nhóm chính: process control, file manipulation, device manipulation, information maintenance, communications, và protection.

### 4.1 Quản lý tiến trình (process control)

- Các Process control system calls bao gồm end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, allocate và free memory.
- Các process phải được tạo, khởi chạy, theo dõi, tạm dừng, tiếp tục và cuối cùng dừng lại.
- Khi một process tạm dừng hoặc dừng, thì một process khác phải được khởi chạy hoặc tiếp tục
- Khi các process dừng bất thường, có thể cần phải cung cấp các lỗi hoặc các công cụ chẩn đoán hoặc phục hồi khác.

### 4.2 Quản lý tệp (file manipulation)

- Các File management system calls bao gồm create file, delete file, open, close, read, write, reposition, get file attributes, và set file attributes.
- Các phương thức này cũng có thể được hỗ trợ cho các thư mục cũng như các tệp thông thường.

### 4.3 Quản lý thiết bị (device manipulation)

- Các Device management system calls bao gồm request device, release device, read, write, reposition, get/set device attributes, và logically attach hoặc detach devices.
- Các thiết bị có thể là vật lý (ví dụ: ổ đĩa) hoặc ảo / trừu tượng (ví dụ: tệp, phân vùng và đĩa RAM).
- Một số hệ thống biểu thị các thiết bị dưới dạng các tệp đặc biệt trong hệ thống tệp, để truy cập "tệp" cần các drivers thiết bị phù hợp trong HĐH. Xem ví dụ thư mục / dev trên bất kỳ hệ thống UNIX nào

#### **4.4 Duy trì thông tin (information maintenance)**

- Các Information maintenance system calls bao gồm calls to get/set the time, date, system data, và process, file, hoặc device attributes.
- Các hệ thống cũng có thể cung cấp khả năng truy xuất bộ nhớ bất cứ lúc nào, các chương trình tạm dừng thực thi sau mỗi lệnh và theo dõi hoạt động của các chương trình, tất cả đều có thể giúp gỡ lỗi chương trình.

#### **4.5 Giao tiếp (communication)**

- Các Communication system calls thực hiện create/delete communication connection, send/receive messages, transfer status information, và attach/detach remote devices
- Cung cấp 2 phương thức: message passing, shared memory và các cơ chế tránh xung đột vùng nhớ

#### **4.6 Bảo vệ (protection)**

- Bảo vệ cung cấp các cơ chế để kiểm soát người dùng / quy trình nào có quyền truy cập vào tài nguyên hệ thống nào
- System call cho phép các cơ chế truy cập được điều chỉnh khi cần thiết và cho người dùng được cấp quyền truy cập nâng cao trong các trường hợp tạm thời được kiểm soát cẩn thận.

### **5. CHƯƠNG TRÌNH HỆ THỐNG**

- Các chương trình hệ thống cung cấp chức năng HĐH thông qua các ứng dụng riêng biệt, không phải là một phần của kernel hoặc command interpreters.
- Hầu hết các hệ thống đi kèm với các ứng dụng hữu ích như máy tính và text editor đơn giản
- Chương trình hệ thống có thể phân chia thành các loại: File management, Status information, File modification, Programming-language support, Program loading và execution, Communications, Background services.

## 6. THIẾT KẾ VÀ TRIỂN KHAI HỆ ĐIỀU HÀNH

### 6.1 Mục tiêu thiết kế

- Xác định yêu cầu các thuộc tính mà hệ thống hoàn thành phải có, và là bước đầu tiên cần thiết trong việc thiết kế bất kỳ hệ thống phức tạp nào. Gồm
- Yêu cầu của người dùng là các tính năng mà người dùng quan tâm và hiểu, và được viết bằng tiếng bản địa.
- Các yêu cầu hệ thống được viết cho các nhà phát triển bao gồm nhiều chi tiết hơn về các chi tiết cụ thể triển khai, yêu cầu về hiệu suất, các ràng buộc tương thích, tuân thủ tiêu chuẩn, v.v.

### 6.2 Chiến lược và cơ chế

- Chiến lược xác định những gì sẽ được thực hiện. Các cơ chế xác định cách thức thực hiện.
- Nếu được phân tách và thực hiện đúng, các thay đổi chiến lược có thể được điều chỉnh dễ dàng mà không cần viết lại mã, chỉ bằng cách điều chỉnh các tham số hoặc có thể tải các tệp dữ liệu / cấu hình mới.

### 6.3 Hiện thực

- Các hệ điều hành truyền thống được viết bằng assembly. Điều này cung cấp sự kiểm soát trực tiếp đối với các vấn đề liên quan đến phần cứng, nhưng gắn chặt một hệ điều hành cụ thể vào một nền tảng cụ thể.
- Gần đây HĐH được viết bằng C, C++. Các đoạn mã quan trọng vẫn được viết bằng assembly
- Hệ điều hành có thể được phát triển bằng cách sử dụng trình giả lập phần cứng đích, đặc biệt là nếu phần cứng thực không khả dụng.

## 7. CẤU TRÚC CỦA HỆ ĐIỀU HÀNH

Để thực hiện hiệu quả, một hệ điều hành nên được phân vùng thành các hệ thống con riêng biệt. Các hệ thống con này sau đó có thể được sắp xếp theo các cấu hình kiến trúc khác nhau:

## 7.1 Kiến trúc đơn giản

- DOS sơ khai là 1 ví dụ. Nó không phân chia hệ thống thành các hệ thống con và không có sự phân biệt giữa chế độ user và kernel, cho phép tất cả các chương trình truy cập trực tiếp vào phần cứng bên dưới.
- Hệ điều hành UNIX ban đầu sử dụng cách tiếp cận phân lớp đơn giản, nhưng hầu như tất cả các hệ điều hành nằm trong một lớp lớn

## 7.2 Phương pháp phân lớp

- Một cách tiếp cận khác là chia hệ điều hành thành một số lớp nhỏ hơn, mỗi lớp nằm trên lớp bên dưới nó và chỉ dựa vào các dịch vụ được cung cấp bởi lớp thấp hơn tiếp theo.
- Cách tiếp cận này cho phép mỗi lớp được phát triển và gỡ lỗi một cách độc lập
- Các cách tiếp cận phân lớp cũng có thể kém hiệu quả hơn, vì một yêu cầu dịch vụ từ lớp cao hơn phải lọc qua tất cả các lớp thấp hơn trước khi nó đạt đến hardware

## 7.3 Microkernel

- Ý tưởng cơ bản đằng sau các hạt vi mô là loại bỏ tất cả các dịch vụ không thiết yếu khỏi kernel và thay vào đó thực hiện chúng như các ứng dụng hệ thống, từ đó làm cho kernel càng nhỏ và hiệu quả càng tốt.
- Bảo mật và bảo vệ có thể được tăng cường, vì hầu hết các dịch vụ được thực hiện trong chế độ người dùng, không phải chế độ kernel.
- Việc mở rộng hệ thống cũng có thể dễ dàng hơn, bởi vì nó chỉ liên quan đến việc thêm nhiều ứng dụng hệ thống, chứ không phải xây dựng lại một kernel mới

## 7.4 Modules

- Phát triển hệ điều hành hiện đại là hướng đối tượng, với nhân lõi tương đối nhỏ và một bộ mô-đun có thể được liên kết động.
- Hạt nhân tương đối nhỏ trong kiến trúc này, tương tự như microkernels, nhưng hạt nhân không phải thực hiện chuyển thông điệp vì các mô-đun có thể tự do liên hệ trực tiếp với nhau.

## 7.5 Hybrid Systems

Hầu hết các hệ điều hành ngày nay không tuân thủ nghiêm ngặt một kiến trúc, là sự kết hợp của một số kiến trúc.

- Kiến trúc Mac OSX dựa trên microkernel Mach cho các dịch vụ quản lý hệ thống cơ bản và kernel BSD cho các dịch vụ bổ sung.
- Hệ điều hành iOS được Apple phát triển cho iPhone và iPad. Nó chạy với ít bộ nhớ và nhu cầu năng lượng điện toán hơn Mac OS X, và hỗ trợ giao diện và đồ họa màn hình cảm ứng cho màn hình nhỏ
- Android bao gồm các phiên bản Linux và máy ảo Java đều được tối ưu hóa cho các nền tảng nhỏ. Các ứng dụng Android được phát triển bằng môi trường phát triển Java - for - Android đặc biệt.

## 8. GỠ LỖI HỆ ĐIỀU HÀNH

Gỡ lỗi bao gồm cả phát hiện, loại bỏ lỗi và điều chỉnh hiệu suất.

### 8.1 Phân tích lỗi

- Trình gỡ lỗi cho phép các quy trình được thực hiện từng bước và cung cấp cho việc kiểm tra các biến và biểu thức khi quá trình thực thi diễn ra.
- Nếu một quá trình thông thường gặp sự cố, kết xuất bộ nhớ trạng thái của bộ nhớ của quá trình đó tại thời điểm xảy ra sự cố có thể được lưu vào một tệp đĩa để phân tích sau.
- Chương trình phải được biên dịch đặc biệt để bao gồm thông tin gỡ lỗi, có thể làm chậm hiệu suất của nó.
- Các cách tiếp cận này không thực sự hoạt động tốt đối với mã hệ điều hành

### 8.2 Điều chỉnh hiệu suất

- Điều chỉnh hiệu suất (gỡ lỗi) yêu cầu giám sát hiệu suất hệ thống.
- Một cách tiếp cận là để hệ thống ghi lại các sự kiện quan trọng vào các tệp nhật ký, sau đó có thể được phân tích bằng các công cụ khác
- Một cách tiếp cận khác là cung cấp các tiện ích sẽ báo cáo trạng thái hệ thống theo yêu cầu, chẳng hạn như lệnh "top" unix. (w, thời gian hoạt động, ps, v.v.)

- Các tiện ích hệ thống có thể cung cấp hỗ trợ giám sát.

### 8.3 DTrace

- DTrace là một cơ sở đặc biệt để truy tìm hệ điều hành đang chạy, được phát triển cho Solaris 10.
- DTrace thêm "probe" trực tiếp vào mã hệ điều hành, có thể được truy vấn bởi "probe consumers".
- Vì DTrace là nguồn mở, nên nó được một số bản phân phối UNIX

## 9. OPERATING-SYSTEM GENERATION

- Các hệ điều hành có thể được thiết kế và xây dựng cho một cấu hình phần cứng cụ thể tại một địa điểm cụ thể, nhưng thông thường hơn, chúng được thiết kế với một số tham số và thành phần biến, sau đó được cấu hình cho một môi trường hoạt động cụ thể.
- Các hệ thống đôi khi cần được cấu hình lại sau khi cài đặt ban đầu, để thêm các tài nguyên, khả năng bổ sung hoặc để điều chỉnh hiệu suất, ghi nhật ký hoặc bảo mật.
- Các thông tin cần thiết bao gồm: CPU, RAM, loại thiết bị, mục đích sử dụng và tính năng yêu cầu
- Mã nguồn HĐH có thể được biên dịch lại và link với một kernel mới
- Khi một hệ thống đã được tạo lại, thường phải khởi động lại hệ thống để kích hoạt kernel mới. Bởi vì có khả năng xảy ra lỗi, hầu hết các hệ thống cung cấp một số cơ chế để khởi động các hạt nhân cũ hơn hoặc thay thế.

## 10. SYSTEM BOOT

- Khi hệ thống bật nguồn, một interrupt được tạo để tải địa chỉ bộ nhớ vào program counter và hệ thống bắt đầu thực hiện các hướng dẫn được tìm thấy tại địa chỉ đó. Địa chỉ này trỏ đến chương trình "bootstrap" nằm trong chip ROM (hoặc chip EPROM) trên bo mạch chủ.
- Chương trình bootstrap ROM trước tiên chạy kiểm tra phần cứng, xác định tài nguyên vật lý nào có mặt và thực hiện tự kiểm tra bật nguồn (POST) của tất cả

các phần cứng. Một số thiết bị, chẳng hạn như controller cards có thể có chẩn đoán trên on-board riêng, được gọi bởi chương trình bootstrap ROM.

- Người dùng thường có tùy chọn nhấn một phím đặc biệt trong quá trình POST, sẽ khởi chạy tiện ích cấu hình ROM BIOS nếu được nhấn. Tiện ích này cho phép người dùng chỉ định và định cấu hình một số tham số phần cứng nhất định để tìm hệ điều hành và có hạn chế quyền truy cập vào tiện ích bằng mật khẩu hay không.
- Đối với hệ thống khởi động đơn, chương trình khởi động được tải khỏi đĩa cứng sau đó sẽ tiến hành xác định vị trí kernel trên ổ cứng, tải kernel vào bộ nhớ và sau đó chuyển điều khiển sang kernel.
- Đối với hệ thống khởi động kép hoặc đa khởi động, chương trình khởi động sẽ cung cấp cho người dùng cơ hội chỉ định một HĐH cụ thể để tải, với lựa chọn mặc định nếu người dùng không chọn một HĐH cụ thể trong khung thời gian nhất định.
- Khi **kernel đang chạy**, nó có thể cho người dùng cơ hội vào chế độ một người dùng, còn được gọi là **chế độ bảo trì**.

## 11. CÂU HỎI

**Câu hỏi 1:** Tại sao các lập trình viên thường không sử dụng trực tiếp system calls mà sử dụng API. Chọn câu trả lời chính xác nhất.

- A. Do người dùng không được phép sử dụng system calls vì đây là các lệnh bên trong kernel
- B. **Do các lệnh system calls thường chi tiết và khó sử dụng, API cung cấp giao diện dễ tiếp cận hơn**
- C. Do system call chỉ chạy được trên môi trường Linux và UNIX. Window không hỗ trợ system call
- D. Tất cả các câu trên đều đúng

**Câu hỏi 2:** Windows Subsystem for Linux 2.0 (WSL2) sử dụng kernel linux mã nguồn mở chính hãng, được biên dịch từ phiên bản 4.19, điều này cho phép truy cập môi trường Linux trên Window. Dựa vào các thông tin trên, hãy chỉ ra phát biểu sai trong các phát biểu sau.

- A. Microsoft đã chuyển sang sử dụng kernel linux và loại bỏ kernel window của họ
- B. WSL2 là một cách cung cấp máy ảo Linux trên Window
- C. WSL2 có thể chạy hầu hết tiện ích của Linux trên máy Window như Docker, câu lệnh terminal,...
- D. Microsoft tùy biến, biên dịch và sử dụng kernel của linux cho mục đích thương mại là việc hợp pháp

Giải thích: D. Do linux là mã nguồn mở, cho phép chỉnh sửa nên việc trên là hợp pháp.

**Câu hỏi 3:** Để giảm sự phụ thuộc vào Android của Google, Huawei đã tung ra Harmony OS (Hongmeng OS) dựa trên kiến trúc Microkernel. OS này hứa hẹn sẽ chạy trên tất cả các thiết bị của Huawei và được cho rằng hiệu suất cao hơn Android nhiều lần. Chọn câu trả lời chính xác nhất về kiến trúc Microkernel.

- A. Microkernel cung cấp tất cả các dịch vụ trong kernel, vì vậy cho hiệu suất cao và có thể chạy trên nhiều phần cứng khác nhau
- B. Việc mở rộng hệ thống dễ dàng hơn kiến trúc monolithic**
- C. Hiệu suất cao nhưng tính bảo mật kém hơn kiến trúc monolithic do hầu hết các dịch vụ được thực hiện trong chế độ kernel
- D. Kích thước kernel lớn hơn kiến trúc monolithic

Giải thích: Bởi vì mở rộng trên kiến trúc microkernel chỉ liên quan đến việc thêm nhiều ứng dụng hệ thống, chứ không phải xây dựng lại một kernel mới như monolithic

**Câu hỏi 4:** Quá trình khởi động máy tuy có thể có các bước khác nhau, tuy nhiên chúng hầu hết thực hiện một số bước cụ thể. Điều nào sau đây sai khi nói về quá trình khởi động máy

- A. Khi hệ thống bật nguồn, một interrupt được tạo để tải địa chỉ bộ nhớ vào program counter
- B. Chương trình bootstrap có nhiệm vụ kiểm tra phần cứng
- C. Đối với hệ thống khởi động kép, người dùng dùng phải chỉ định một HĐH để chạy, nếu không máy sẽ chờ liên tục**

D. Đối với hệ thống khởi động đơn, chương trình khởi động được tải khỏi đĩa cứng sau đó sẽ tiến hành xác định vị trí kernel trên ổ cứng.

**Giải thích:** Nếu người dùng không chọn, HĐH mặc định sẽ được tải trong khung thời gian nhất định.

**Câu 5:** Linux là một trong những mã nguồn HĐH mở phổ biến nhất thế giới, nổi tiếng nhất là các bản distro như Ubuntu, CentOS, Kali Linux,... Chỉ riêng kernel Linux đã tốn 27.8 triệu dòng code, bạn hãy cho biết mã nguồn kernel Linux được viết bằng ngôn ngữ nào?

- A. C++ và assembly
- B. C và assembly
- C. C, C++ và assembly
- D. Chỉ viết bằng C

**Giải thích:** Phần lớn được viết bằng C, có 1 số nhỏ được viết bằng ngôn ngữ assembly có đuôi .s

**Câu 6:** Trên linux, ta có thể kiểm quyền truy cập file bằng câu lệnh:

**ls -l /duong-dan-file-hoac-folder**

Sau khi chạy câu lệnh trên, ta thu được chuỗi rw- ứng với quyền của Guest, hãy chọn đáp án đúng về chuỗi này.

- A. Guest không thể đọc file này
- B. Chuỗi rw- có giá trị nhị phân là 101 tương đương 5 trong thập phân
- C. Guest không thể thực thi file này
- D. Ta không thể thêm quyền cho Guest

**Câu 7:** Hệ điều hành là một phần không thể thiếu trong các hệ thống máy tính cá nhân, nó cung cấp các dịch vụ cần thiết cho người dùng. Điều này sau đây không phải nhiệm vụ của HĐH ?

- A. Cung cấp giao diện tương tác với người dùng (VD như GUI, C,...)
- B. Cung cấp driver cho các thiết bị ngoại vi**
- C. Định thời CPU và quản lý tiến trình

#### D. Quản lý file và tệp tin

Giải thích: Nhà sản xuất thiết bị ngoại vi sẽ cung cấp driver cho thiết bị đó.

# CHƯƠNG 3: PROCESS

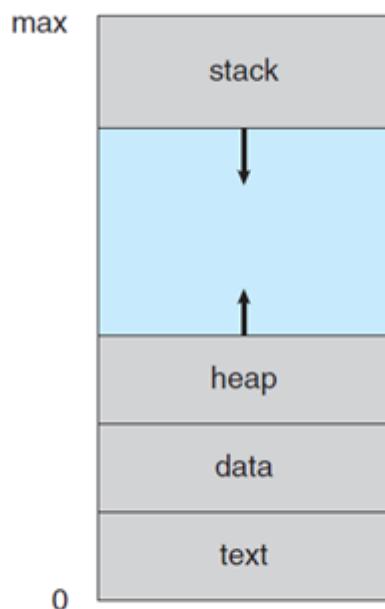
## 3.1 Khái niệm process

### 3.1.1 Process

Process là một chương trình đang thực thi. Nó cung cấp khả năng các hoạt động diễn xẩy đồng thời ngay cả khi máy tính chỉ có 1 CPU.

Memory layout của một process thường chia thành các phần chính sau:

- Text section: vùng chứa code thực thi.
- Data section: vùng chứa biến toàn cục.
- Heap section: vùng được cấp phát động trong khi chương trình đang thực thi.
- Stack section: vùng chứa biến cục bộ, tham số của hàm, địa chỉ trả về,...



**Figure 3.1** Layout of a process in memory.

Lưu ý: **kích thước của text section và data section được cố định, còn kích thước stack section và heap section có thể co giãn** trong khi chương trình đang thực thi.

Phân biệt khái niệm “program” và “process”:

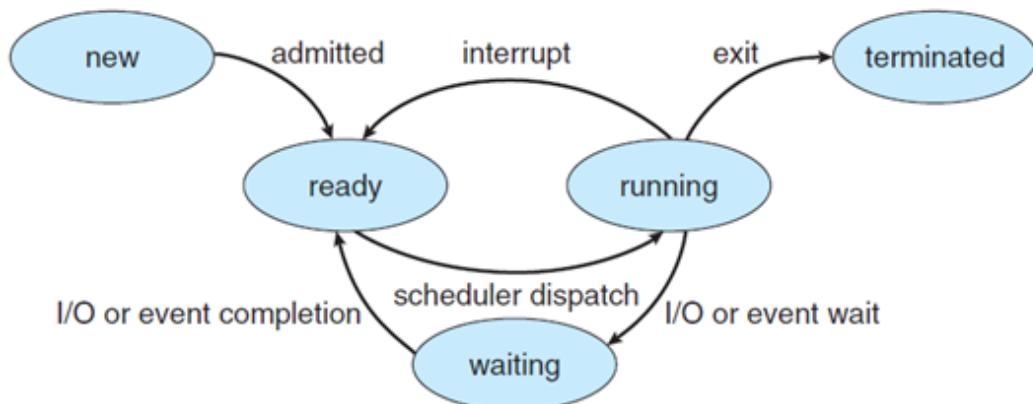
- **Program:** là một thực thể **thụ động**, bao gồm các câu lệnh được lưu trong đĩa.
- **Process:** là một thực thể **chủ động**, với một bộ đếm chương trình, định rõ câu lệnh tiếp theo được thực thi và có các tài nguyên liên quan. Một program trở thành một process khi nó được tải vào bộ nhớ để thực thi.

Một điều đặc biệt đối với process là nếu **2 processes cùng liên kết với một chương trình, thì chúng vẫn được xem như 2 luồng thực thi riêng rẽ.**

### 3.1.2 Trạng thái process

Khi một process đang thực thi, nó thay đổi trạng thái của nó. Một process có thể có các trạng thái sau:

- New: process được tạo ra
- Running: các câu lệnh đang được thực thi.
- Waiting: process đang đợi một số sự kiện xảy ra (yêu cầu về I/O,...)
- Ready: process sẵn sàng được gán cho bộ xử lí để được thực thi.
- Terminated: process kết thúc quá trình thực thi.



**Figure 3.2** Diagram of process state.

### 3.1.3 Process control block

Mỗi process được biểu diễn trong hệ điều hành bởi một khối điều khiển process (Process control block – PCB), nó bao gồm các thông tin liên quan đến một process.

- Process state: bao gồm các giá trị new, ready, running, waiting,...
- Program counter: bộ đếm lưu trữ địa chỉ của lệnh sẽ được thực thi tiếp theo.
- CPU registers: bao gồm con trỏ stack, chỉ mục các thanh ghi, các thanh ghi chung,...
- CPU-scheduling information: bao gồm các thông tin về ưu tiên process, con trỏ đến hàng đợi process,...
- Memory-management information: bao gồm giá trị cơ bản và giới hạn của các thanh ghi,...
- Accounting information: bao gồm thông tin về phần trăm sử dụng CPU, thời gian thực sử dụng,...
- I/O status information: bao gồm thông tin về các thiết bị I/O được cấp phát cho process,...

## 3.2 Process Scheduling

Mục đích của đa chương trình là có nhiều process cùng chạy cùng lúc để tối ưu hóa CPU. Số lượng process hiện tại trong bộ nhớ được gọi là bậc của đa chương trình.

### 3.2.1 Scheduling Queues

Khi một process đi vào hệ thống, nó được đặt trong hàng đợi sẵn sàng (ready queue), chờ để được cấp phát core để thực thi. Hàng đợi này thường được lưu dưới dạng danh sách liên kết.

Khi một process yêu cầu sự kiện xảy ra thì sẽ được đặt trong hàng đợi chờ. (wait queue).

Có nhiều trường hợp một process được đặt vào hàng đợi chờ:

- Khi một process đưa ra yêu cầu I/O
- Khi một process tạo ra các processes con và đợi processes con kết thúc

- Khi một process bị bắt buộc rời khỏi core, là kết quả của một gián đoạn.

### 3.2.2 CPU Scheduling

Một process thay đổi vị trí ở hàng đợi sẵn sàng và hàng đợi chờ trong suốt quá trình thực thi. Vai trò của bộ định thời CPU là chọn ra process trong hàng đợi sẵn sàng để cấp phát core CPU cho nó.

### 3.2.3 Context Switch

Một interrupts (ngắt) dẫn đến hệ điều chuyển CPU core từ nhiệm vụ hiện tại và chạy các tác vụ ở kernel. Khi điều đó xảy ra, hệ thống cần lưu lại ngữ cảnh của process đang chạy trên CPU core để nó có thể phục hồi lại ngữ cảnh đó khi các tác vụ ở kernel đã hoàn tất.

Hệ thống sẽ làm việc không tốt khi sự chuyển ngữ cảnh xảy ra. Thời gian chuyển ngữ cảnh phụ thuộc nhiều vào sự hỗ trợ phần cứng.

## 3.3 Các hoạt động trên process

Trong hầu hết các hệ thống, các process có thể thực hiện đồng thời, và chúng có thể được tạo ra và xóa linh động. Vì vậy, hệ thống phải cung cấp một nguyên lý cho việc xử lí các vấn đề này.

### 3.3.1 Tạo process

Hầu hết các hệ thống định danh process theo một số tự nhiên duy nhất, đó là process identifier (PID). Nó được sử dụng như một chỉ mục để hệ thống có thể truy cập đến các tính chất của nó trong kernel.

Thông thường, khi một process tạo một process con thì process này sẽ cần một số tài nguyên nhất định để hoàn thành nhiệm vụ này. Process con sau đó có thể nhận được tài nguyên trực tiếp từ hệ điều hành hoặc từ tài nguyên của process cha.

Để tạo một process con, ta dùng lệnh **fork()**. Giá trị trả về của hàm này đối với process cha là PID của process con, và đối với process con là 0. Nếu xảy ra lỗi, giá trị trả về sẽ là một số âm.

Khi một process cha tạo một process con, có 2 trường hợp về thực thi sau xảy ra:

- Process cha tiếp tục thực thi đồng thời với process con.
- Process cha đợi cho đến khi một số hoặc tất cả các process con kết thúc thì nó mới thực thi tiếp.

Cũng có 2 trường hợp về không gian địa chỉ của process con:

- Process con có chương trình và dữ liệu giống với process cha.
- Process con có một chương trình mới được truyền vào cho nó.

Hình sau tóm tắt quá trình tạo process mới:

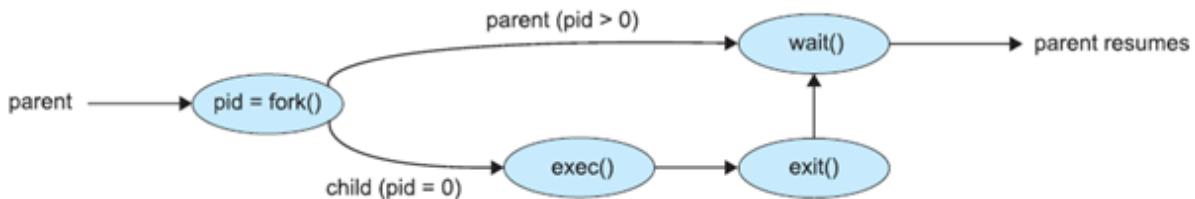


Figure 3.9 Process creation using the `fork()` system call.

### 3.3.2 Kết thúc process

Một process kết thúc khi nó hoàn thành việc thực thi câu lệnh cuối cùng và yêu cầu hệ điều hành xóa nó bằng lời gọi hệ thống `exit()`. Lúc này, process có thể trả về giá trị trạng thái của nó cho process cha thông qua lời gọi hệ thống `wait()`.

```
pid_t pid;  
int status;  
pid = wait(&status);
```

Khi một process kết thúc, nhưng cha của nó chưa gọi `wait()` thì nó được gọi là **zombie process**. Tất cả process đều qua trạng thái này khi chúng kết thúc.

Khi cha của process không gọi `wait()` mà kết thúc luôn thì process con được gọi là **orphans**.

### 3.4 Giao tiếp giữa các process

Khi các process thực thi, chúng có thể thực thi độc lập hoặc có hợp tác với nhau:

- **Thực thi độc lập:** process không chia sẻ dữ liệu với các process khác trong hệ thống.
- **Thực thi hợp tác:** process có ảnh hưởng hoặc bị ảnh hưởng bởi process khác trong hệ thống.

Có một vài lí do cho việc cung cấp môi trường cho phép các process thực thi hợp tác với nhau:

- **Chia sẻ thông tin:** một vài ứng dụng có thể dùng chung đến một phần thông tin.
- **Tăng tốc độ tính toán:** chia nhỏ một nhiệm vụ thành các nhiệm vụ nhỏ hơn để chạy song song với nhau. (trong trường hợp máy tính có nhiều CPU core).
- **Module hóa:** chúng ta có thể xây dựng hệ thống theo hướng nhiều module chức năng.

Sự hợp tác của các process yêu cầu một phương thức **interprocess communication (IPC)** để cho phép chúng trao đổi dữ liệu. Có hai mô hình cơ bản của IPC là **shared memory** (chia sẻ bộ nhớ) và **message passing** (truyền tin).

### 3.5 Giao tiếp giữa các process trong hệ thống sử dụng vùng nhớ chia sẻ

IPC sử dụng shared memory để hình thành nên một vùng chia sẻ bộ nhớ. Thông thường, vùng bộ nhớ được chia sẻ nằm trong không gian địa chỉ của **process tạo nên nó**. Các process khác muốn giao tiếp mà sử dụng vùng nhớ này phải liên kết nó vào không gian địa chỉ của mình. Những process này chịu trách nhiệm về việc đảm bảo rằng chúng **không ghi đồng thời lên vùng một vùng địa chỉ**.

Để làm rõ hơn vấn đề này, hãy cùng xem xét vấn đề **producer – consumer**. Một producer process tạo ra thông tin được tiêu thụ bởi một consumer process. **Ví dụ, trình biên dịch có thể tạo ra code hợp ngữ, sản phẩm này sẽ được tiêu thụ bởi trình hợp ngữ.** Tiếp đó, **trình hợp ngữ tạo ra mã máy, sản phẩm này lại được tiêu thụ bởi loader.** Vấn đề producer – consumer còn có

liên quan đến mô hình client – server, trong đó server đóng vai trò là producer và client đóng vai trò là consumer.

Chúng ta có thể dùng phương pháp chia sẻ vùng nhớ để giải quyết vấn đề producer – consumer.

Mô tả code để giải quyết vấn đề. Buffer trống khi  $in == out$  và đầy khi  $(in + 1) \% BUFFER\_SIZE == out$ .

```
#define BUFFER_SIZE 10

typedef struct {

    . . .

} item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;
```

Cách thức này cho phép có tối đa trong BUFFER cùng lúc có BUFFER – 1 items. Code cho producer process và consumer process.

```
item next_produced;

while (true) {

    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)

        ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;

}
```

```
item next_consumed;

while (true) {

    while (in == out)

        ; /* do nothing */

    next_consumed = buffer[out];

    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_consumed */

}
```

### 3.6 Giao tiếp giữa các process trong hệ thống truyền thông điệp

Thông thường có 2 loại thông điệp được gửi đi và nhận lại trong quá trình giao tiếp giữa các process:

- Thông điệp cố định dung lượng: dễ dàng về mặt hiện thực ở cấp độ hệ thống nhưng khó khăn cho người lập trình.
- Thông điệp biến đổi về dung lượng: cần hiện thực ở mức độ hệ thống phức tạp, nhưng dễ dàng hơn cho người lập trình.

Sự đánh đổi giữa hai điều này là thường thấy ở việc thiết kế hệ điều hành.

Khi hai process P và Q muốn giao tiếp, một **liên kết giao tiếp (communication link)** sẽ tồn tại giữa chúng. Ở đây, chúng ta đề cập đến việc hiện thực liên kết theo mặt logic và các tác vụ **send()/receive()**.

- Giao tiếp có hướng hoặc không hướng.
- Giao tiếp đồng bộ hoặc bất đồng bộ.
- Bộ nhớ đệm tự động hay không.

#### 3.6.1 Naming

Processes muốn giao tiếp với nhau phải có cách để liên kết được với nhau, chúng có thể sử dụng giao tiếp trực tiếp hoặc gián tiếp.

Về giao tiếp trực tiếp, mỗi process muốn giao tiếp phải chỉ rõ tên của người nhận hay người gửi, phương thức send() và receive() được định nghĩa:

- send(P, message) – gửi thông điệp đến process P
- receive (Q, message) – nhận thông điệp từ process Q

Tính chất:

- Liên kết sẽ được hình thành tự động giữa các cặp process cần giao tiếp.
- Liên kết hình thành chỉ kết nối đúng 2 process với nhau.
- Giữa 2 process chỉ có một liên kết được hình thành.

### 3.6.2 Bất đồng bộ

Thông điệp gửi đi có thể bị khóa (blocking) hoặc không bị khóa (nonblocking) – còn được xem là đồng bộ (synchronous) và bất đồng bộ (asynchronous).

- **Blocking send:** process gửi thông điệp bị khóa cho đến khi tin nhắn được nhận bởi process nhận hoặc hòm thư.
- **Nonblocking send:** process gửi thông điệp gửi thông điệp và tiếp tục thực thi.
- **Blocking receive:** process nhận thông điệp bị khóa cho đến khi thông điệp đã có sẵn.
- **Nonblocking receive:** process nhận thông điệp nhận thông điệp hợp lệ hoặc null.

### 3.6.3 Buffering

Cho dù sự giao tiếp là trực tiếp hay gián tiếp thì những thông điệp trao đổi giữa các process sẽ ở trong một **hàng đợi (queue)** tạm thời. Về cơ bản, những hàng đợi này có thể được hiện thực theo 3 cách:

- **Zero capacity (sức chứa 0):** hàng đợi có độ dài tối đa bằng 0, vì vậy liên kết không chứa bất kỳ tin nhắn nào. Trong trường hợp này, process gửi thông điệp phải bị block cho đến khi process nhận nhận được thông điệp.
- **Bounded capacity (sức chứa có giới hạn):** hàng đợi có chiều dài giới hạn n, vì vậy sẽ có tối đa n thông điệp có thể đợi. Một thông điệp có thể đợi ở hàng đợi nếu hàng đợi đó chưa đầy và process gửi thông điệp có thể tiếp tục thực thi. Nếu hàng đợi đã đầy thì process gửi thông điệp sẽ bị block cho đến khi có không gian trống trong hàng đợi.

- **Unbounded capacity:** độ hàng của dài đợi không giới hạn, bất cứ thông điệp nào cũng có thể tham gia chờ và process gửi thông điệp không bao giờ bị block.

### 3.7 Ví dụ hệ thống giao tiếp process

#### 3.7.1 POSIX Shared Memory

Một vài nguyên lý IPC có sẵn cho hệ thống POSIX (tiêu chuẩn hệ điều hành di động), trong đó có “chia sẻ vùng nhớ” (shared memory).

POSIX shared memory được tổ chức trên việc sử dụng những file memory-mapped, tức là liên kết những vùng nhớ được chia sẻ với một file.

#### 3.7.2 Mach Message Passing

Một ví dụ về “message passing” ở trong hệ điều hành Mach. Mach được thiết kế đặc biệt cho hệ thống phân tán, nhưng cũng phù hợp với hệ thống máy tính và di động. Một số lớn lượng giao tiếp trong Mach được thực hiện thông qua “thông điệp”. Thông điệp được gửi, nhận từ hòm thư, còn được gọi là “cổng”. Cổng có giới hạn về kích thước và một chiều. Thông điệp được gửi từ một cổng và sự phản hồi được gửi ở một cổng reply khác. Mỗi cổng có thể có nhiều process gửi, nhưng chỉ có một process nhận.

#### 3.7.3 Window

Hệ điều hành Windows là một điển hình của mô hình thiết kế hiện đại, cho phép tăng các hàm chức năng và giảm thời gian hiện thực. Window cung cấp hỗ trợ trên nhiều môi trường hoặc hệ thống con. Các chương trình ứng dụng giao tiếp với hệ thống con thông qua cơ chế truyền tin nhắn (message-passing).

Message-passing trên Windows được gọi là "thủ tục cục bộ nâng cao" (advanced local procedure call - ALPC). Nó được sử dụng để giao tiếp giữa hai quá trình trên cùng thiết bị. Tương tự như "cuộc gọi thủ tục từ xa" (remote procedure call - RPC) nhưng được tối ưu hóa và dành riêng cho Windows. Giống như Mach, Windows dùng các cổng để kết nối hai

quá trình. Windows sử dụng 2 loại cổng là cổng kết nối (connection port) và cổng giao tiếp (communicate port).

Máy chủ xử lý các cổng kết nối dùng chung - phần được nhìn thấy bởi tất cả các quá trình. Khi người dùng muốn yêu cầu từ hệ thống con, hệ thống này sẽ yêu cầu cổng kết nối của máy chủ và gửi yêu cầu đến cổng đó. Máy chủ sẽ tạo 1 kênh truyền và trả về trình xử lý cho người dùng. Kênh này bao gồm 1 cặp giao tiếp riêng tư, 1 cho tin nhắn người dùng - hệ thống (client-server message), 1 cho tin nhắn hệ thống - người dùng (server-client message). Ngoài ra các kênh giao tiếp cũng hỗ trợ cơ chế phản hồi (callback mechanism) cho phép người dùng và hệ thống chấp nhận yêu cầu nếu muốn nhận được phản hồi.

Khi kênh ALPC được tạo ra, một trong ba kỹ thuật truyền tin nhắn sau sẽ được thông qua:

- Đối với tin nhắn nhỏ (<=256 bytes), cổng hàng đợi tin nhắn sẽ được chọn như bộ nhớ trung gian và tin nhắn được sao chép từ quá trình này sang quá trình khác.
- Những tin nhắn lớn hơn phải được truyền thông qua một "đoạn cắt" - là một đoạn nhớ được chia sẻ với kênh.
- Khi lượng dữ liệu quá lớn để khớp với một đoạn cắt, một API có sẵn sẽ cho phép hệ thống đọc và ghi trực tiếp vào địa chỉ làm việc của người dùng

Người dùng phải quyết định khi nào nên cài đặt kênh, liệu có cần thiết phải gửi một tin nhắn lớn hay không. Nếu người dùng muốn gửi tin nhắn lớn, yêu cầu phải có một đoạn cắt được tạ ra để nó có thể được sử dụng.

Cần phải chú ý rằng "thủ tục nội bộ nâng cao" (ALPC) trên Windows không phải là một phần của API. Hơn nữa những ứng dụng sử dụng Windows API sẽ gọi những "cuộc gọi thủ tục từ xa" - RPC. Khi RPC được gọi bởi một quá trình trên cùng hệ thống, RPC sẽ xử lý gián tiếp thông qua lời gọi thủ tục ALPC.

### 3.7.4 Pipes

#### 3.7.4.1 Ordinary pipes

Các pipes thông thường cho phép 2 quá trình giao tiếp với nhau theo kiểu producer - consumer: producer ghi vào 1 điểm kết thúc của pipe (write end) và consumer đọc từ một điểm kết thúc khác (read end). Vì vậy, pipes là đơn hướng, chỉ cho phép giao tiếp một chiều. Nếu giao tiếp 2 chiều được yêu cầu, 2 pipe phải được sử dụng với mỗi pipe truyền dữ liệu theo 1 hướng khác nhau.

Trên UNIX, pipes thông thường được khởi tạo bằng:

```
pipe(int fd[])
```

Hàm khởi tạo pipe được truy cập thông qua int fd[], mô tả file fd[0] là phần read end của pipe, fd[1] là write end. UNIX xử lý pipe như một loại file đặc biệt. Vì vậy, pipe có thể được truy cập bằng cách sử dụng các system call **read()** và **write()**.

Pipe thông thường không thể được truy cập từ bên ngoài quá trình tạo ra nó. Thông thường, quá trình cha tạo ra một pipe và sử dụng nó để giao tiếp với các quá trình con mà nó tạo ra bằng **fork()**.

Pipe thông thường trên Windows được gọi là anonymous pipe, và cũng tương tự như trên UNIX, chúng một chiều và sử dụng quan hệ cha - con trong giao tiếp của các quá trình. Hơn nữa, đọc và ghi pipe có thể được hiện thực với hàm **ReadFile()** và **WriteFile()**. Windows API để tạo pipe là **CreatePipe()**, bao gồm 4 tham số bao gồm đọc, ghi vào pipe, kiến trúc **STARTUPINFO** và kích thước của pipe (byte).

#### 3.7.4.2 Named pipe

Pipe thông thường cung cấp một cơ chế đơn giản cho một cặp quá trình giao tiếp với nhau. Tuy nhiên, pipe thông thường chỉ tồn tại khi một quá trình đang giao tiếp với quá trình khác. Trên cả UNIX và Windows một khi quá trình kết thúc thì pipe thông thường cũng biến mất.

Pipe được đặt tên (Named pipe) cung cấp một công cụ giao tiếp mạnh mẽ hơn. Giao tiếp 2 chiều và không yêu cầu quan hệ cha con của quá trình. Một khi named pipe được tạo ra, nhiều quá trình có thể sử dụng nó để giao tiếp. Trên thực tế, trong một vài hoàn cảnh đặc biệt, 1 named pipe có nhiều bộ ghi. Hơn nữa named pipe vẫn tồn tại sau khi giao tiếp giữa các quá trình kết thúc. Cả UNIX và Windows đều hỗ trợ named pipe nhưng hiện thực khác nhau.

Named pipe sử dụng FIFO trên UNIX. Một khi được tạo ra, chúng sẽ tồn tại như một file đặc trưng trong hệ thống. Một hàng đợi được tạo bởi mkfifo() và được điều khiển bởi các system call **open()**, **read()**, **write()** và **close()**. Chúng sẽ vẫn tồn tại đến khi nào bị xóa khỏi hệ thống file. Mặc dù FIFO cho phép giao tiếp 2 chiều, tuy nhiên chỉ những truyền dẫn bán song được cho phép. Nếu muốn dữ liệu được truyền song song, 2 FIFO phải được sử dụng. Ngoài ra, những quá trình có giao tiếp phải ở cùng một thiết bị. Nếu muốn giao tiếp liên thiết bị sockets phải được sử dụng.

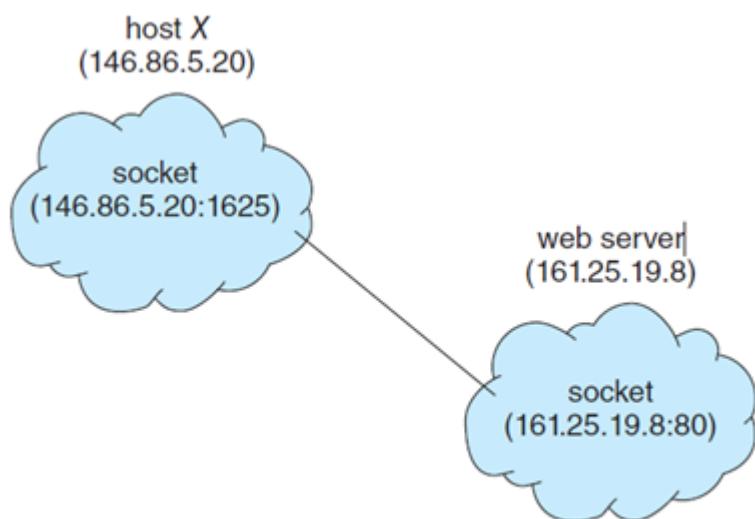
Named pipe trên Windows cung cấp phương thức giao tiếp rộng hơn trên UNIX. Giao tiếp song song hoàn toàn được cho phép, và các quá trình có thể trên cùng hoặc khác thiết bị với nhau. Hơn nữa, chỉ những dữ liệu hướng byte mới có thể truyền thông qua UNIX FIFO nhưng trên Windows, hệ thống chấp nhận cả dữ liệu hướng byte hoặc lời nhắn. Named pipe được tạo thông qua hàm CreateNamedPipe(), và người dùng có thể kết nối với named pipe bằng **ConnectNamedPipe()**. Giao tiếp qua named pipe có thể được hiện thực nhờ **ReadFile()** và **WriteFile()**.

### 3.8 Giao tiếp giữa hệ thống Client-Server

Những kỹ thuật như “shared memory” và “message passing” có thể được sử dụng trong việc giao tiếp giữa hệ thống client-server. Ngoài ra, có thêm các kỹ thuật khác như **sockets** và **remote procedure calls (RPCs)**.

### 3.8.1 Sockets

Một **socket** được định nghĩa như là một điểm kết thúc của sự giao tiếp. Sự giao tiếp giữa hai process thông qua network sử dụng 2 socket, mỗi socket cho mỗi process. Một socket được định danh bởi sự kết hợp của địa chỉ IP và số cổng. Thông thường, socket sử dụng kiến trúc client-server. Server đợi các yêu cầu của client bằng cách theo dõi một số cổng xác định. Một khi yêu cầu được nhận, server chấp nhận kết nối từ socket của client để hoàn tất kết nối. Khi một client process khởi tạo một yêu cầu, nó được gán cho cổng bởi máy tính chủ của nó. Cổng này có một số bất kỳ lớn hơn 1024. Ví dụ, nếu client ở máy X có địa chỉ IP 146.86.5.20 muốn tạo một kết nối với web server (đang theo dõi cổng 80) tại địa chỉ 161.25.19.8, máy X có thể được gán cho cổng 1625. Như hình dưới:



**Figure 3.26** Communication using sockets.

Những gói tin sẽ được chuyển đến các process tương ứng với số của cổng đến.

Tất cả các kết nối phải là duy nhất, vì vậy nếu process khác trên máy X muốn tạo 1 kết nối khác với web server như thì nó sẽ được gán cho một cổng khác có số lớn hơn 1024 và khác 1625.

### 3.8.2 Thủ tục lời gọi từ xa

Một trong những hình thức của dịch vụ từ xa (remote service) là mẫu RPC, được thiết kế như một cách để trùu tượng hóa nguyên lý procedure-call được sử dụng giữa các hệ thống kết nối thông qua mạng. Vì chúng ta đang xử lí các process chạy trên các hệ thống khác nhau nên phải sử dụng mô hình giao tiếp dựa trên thông điệp.

Những thông điệp được trao đổi trong giao tiếp RPC phải có cấu trúc tốt và vì vậy không chỉ đơn thuần là các gói dữ liệu. Mỗi thông điệp được gửi đến một **RPC daemon** đang theo dõi một cổng trong hệ thống từ xa, và mỗi thông điệp chứa một định danh cho hàm thực thi và tham số được truyền vào hàm đó. Hàm này sau đó được thực thi như yêu cầu, và kết quả được gửi về cho process gửi thông điệp thông qua một thông điệp riêng.

## 3.9 Câu hỏi trắc nghiệm

3.9.1 Vùng nhớ của một process được chia thành các vùng nào nhằm nâng cao hiệu quả?

- A. Text section, data section, heap, stack
- B. Data section, heap, stack, virtual
- C. Text section, data section, stack, unmapped region
- D. Text section, heap, stack, mapped region

3.9.2 Cho đoạn chương trình sau:

```
pid_t a;  
  
int b;  
  
a = wait(&b);
```

Giá trị nào là giá trị kết thúc của process con?

- A. a
- B. b
- C. A và B sai
- D. A và B đúng

3.9.3 Môi trường cho phép sự giao tiếp giữa các process có ưu điểm gì?

- A. Chia sẻ được các thông tin chung
- B. Tăng tốc độ hệ thống

C. Cấu trúc hệ thống theo module tốt hơn.

**D. Cả 3 phương án trên**

3.9.4 Trong hệ điều hành Linux, lệnh nào sau đây được dùng để tạo một process con?

A. CreateProcess

B. NewProcess

**C. fork**

D. ForkProcess

3.9.5 Vùng stack có chứa địa chỉ của loại biến nào sau đây:

A. Biến toàn cục

**B. Biến con trỏ**

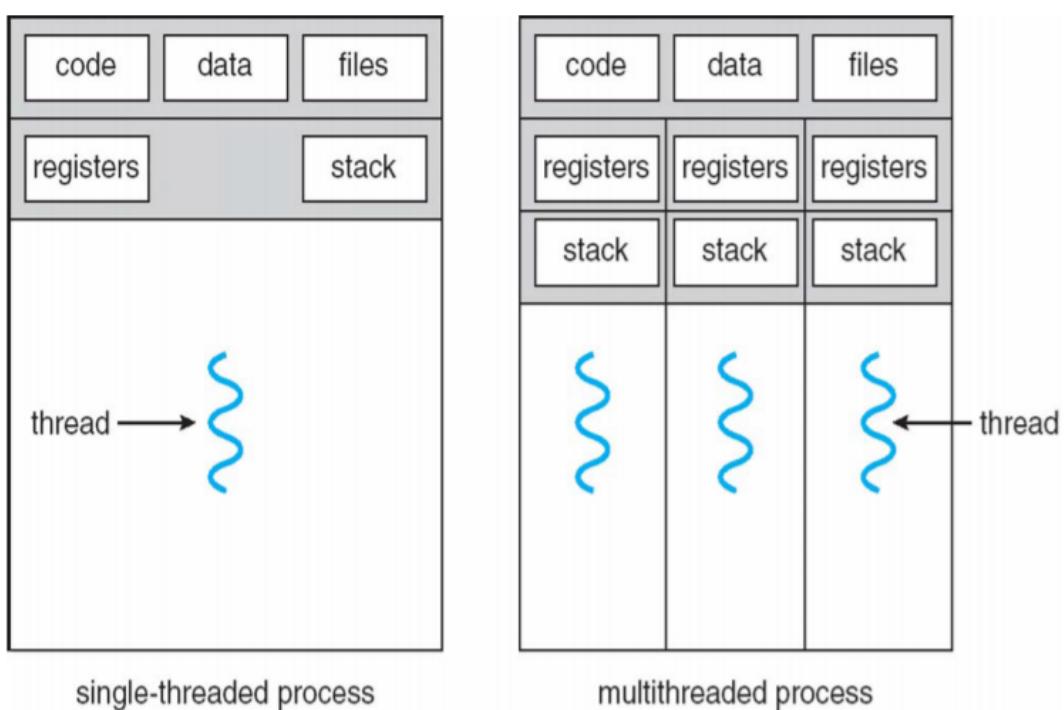
C. Biến pid của một process

D. Biến static

# CHƯƠNG 4: LUỒNG & THỰC THI ĐỒNG THỜI

## 1. Tổng quan

- **Luồng (thread)** là đơn vị thực thi cơ bản của CPU, nó bao gồm: **thread ID, program counter (PC), các register, và stack.**
- Các luồng thực thi trong 1 quá trình (process) **chia sẻ với nhau về vùng nhớ code, vùng nhớ data và các tài nguyên khác của hệ điều hành.**
- Mở rộng khái niệm quá trình truyền thống bằng cách hiện thực nhiều luồng thực thi trong cùng một môi trường của quá trình, gọi là quá trình đa luồng (**multithreaded process**).



## Hình 1.1: So sánh giữa quá trình đơn luồng và đa luồng

### 1.1 Tại sao nên đa luồng (multithreaded)?

- Hầu hết mọi phần mềm hiện nay đều thực thi đa luồng.
- Quá trình đa nhiệm có thể hiện thực bằng nhiều thread riêng biệt.
- Việc tạo ra process tốn tài nguyên và nặng hơn việc tạo ra thread.
- Code đơn giản và tăng hiệu quả.
- Đa phần kernel đều thực thi đa luồng.

### 1.2 Lợi ích của multithreaded

- 1) **Tính đáp ứng (Responsiveness):** cao cho các ứng dụng tương tác multithreaded
- 2) **Chia sẻ tài nguyên (Resource sharing):** ví dụ memory
- 3) **Tính kinh tế (Economy):** Tiết kiệm chi phí hệ thống
  - a) Chi phí tạo/quản lý thread nhỏ hơn so với quá trình
  - b) Chi phí chuyển ngữ cảnh giữa các thread nhỏ hơn so với quá trình
- 4) **Khả năng mở rộng (Scalability):** tận dụng được đa xử lý
  - a) Mỗi thread chạy trên một processor riêng, do đó tăng mức độ song song của chương trình.

## 2. Lập trình xử lý đa lõi (Multicore Programming)

- Một hệ thống đa nhân (Multi-core processor) có thể thực thi đồng thời với vài luồng thực thi song song nhau do hệ thống có thể phân chia những luồng thực thi cho các nhân khác nhau.
- Sự khác biệt giữa thực thi đồng thời (**Concurrency**) và thực thi song song (**Parallelism**):
  - Thực thi đồng thời (Concurrency) hỗ trợ thực thi đa tác vụ bằng cách tạo một tiến trình thực thi tác vụ (có thể tuần tự hoặc song song).
  - Thực thi song song (Parallelism) thực hiện nhiều tác vụ cùng một lúc.
  - Vì vậy có thể có thực thi đồng thời nhưng không song song.

### 2.1 Các vấn đề của việc lập trình trên hệ thống đa lõi

1. **Phân chia tác vụ (Identifying tasks).**
2. **Cân bằng (Balance):** khi phân chia tác vụ phải đảm bảo mỗi tác vụ thực hiện

số lượng công việc tương đương nhau.

3. **Phân tách dữ liệu (Data splitting):** việc truy cập và thay đổi dữ liệu phải được chia để chạy trên các lõi riêng biệt.
4. **Tính độc lập dữ liệu (Data dependency):** khi tác vụ có phụ thuộc dữ liệu với tác vụ khác, phải kiểm tra tính đồng bộ hóa để đảm bảo sự phụ thuộc dữ liệu
5. **Kiểm tra và gỡ lỗi (Testing and debugging):** khó hơn so với ứng dụng thực thi đơn luồng.

## 2.2 Định luật Amdahl

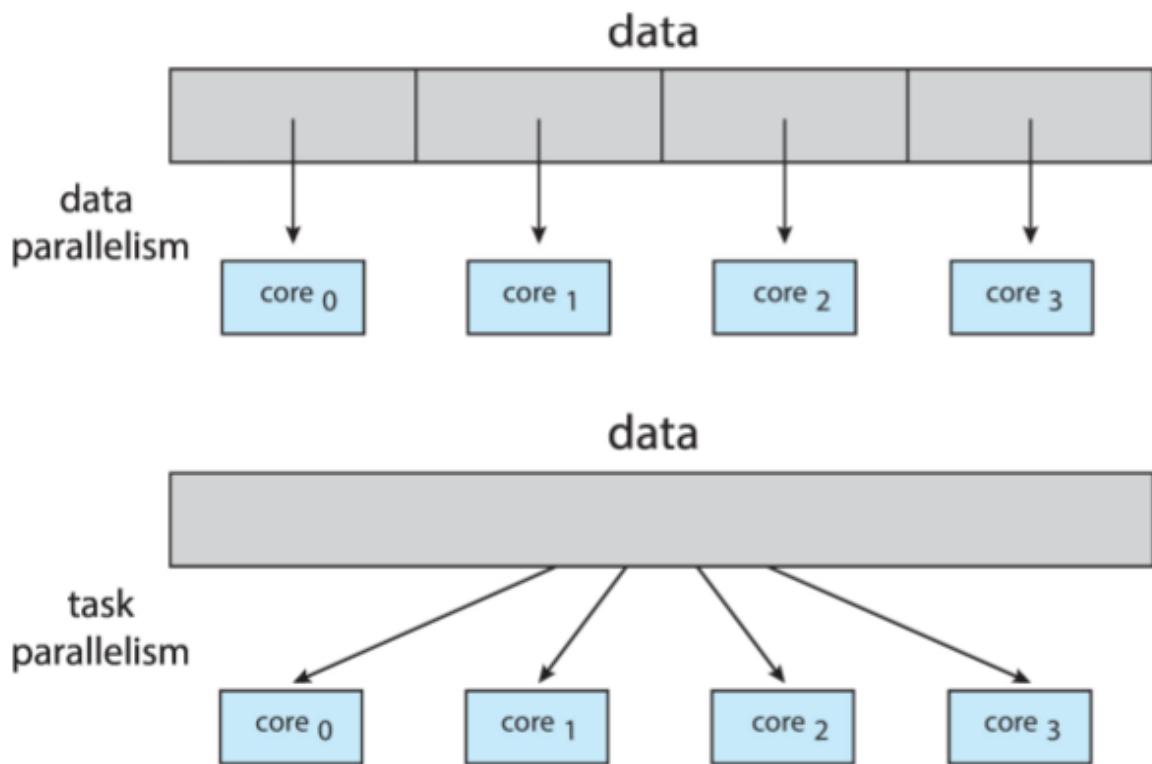
Tính toán độ tăng tốc hiệu suất dựa trên việc bổ sung thêm lõi cho ứng dụng vừa có những thành phần xử lý tuần tự và song song.

$$\text{Độ tăng tốc: } speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- S: tỉ lệ phần tuần tự
- N: Số lõi xử lý

Khi N cực lớn thì độ tăng tốc xấp xỉ  $1/S$ . Vì vậy các tỉ lệ xử lý tuần tự càng lớn thì tốc độ tăng tốc hiệu suất dựa trên việc bổ sung thêm lõi càng nhỏ và ngược lại.

## 2.3 Các loại xử lý song song



Hình 2.3: So sánh hai loại xử lý song song.

### 2.3.1 Xử lý song song về dữ liệu (Data parallelism)

- Phân bổ những phần dữ liệu của cùng một dữ liệu trên nhiều lõi xử lý và thực hiện cùng một nhiệm vụ trên mỗi lõi.
- Sử dụng trong những bài toán lớn, có thể chia nhỏ và tổng hợp lại.

### 2.3.2 Xử lý song song về nhiệm vụ (Task parallelism)

- Phân bổ nhiệm vụ khác nhau cho từng luồng. Mỗi luồng thực hiện từng nhiệm vụ riêng biệt.
- Các luồng khác nhau có thể thực hiện tác vụ trên cùng một dữ liệu hoặc trên những dữ liệu khác nhau.
- Một ứng dụng có thể kết hợp cả hai loại xử lý song song

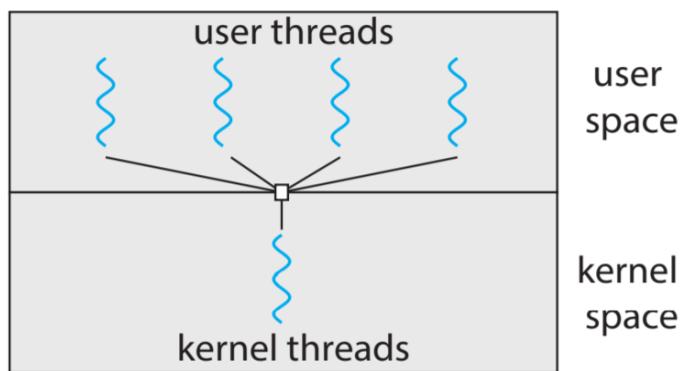
## 3. Các mô hình xử lý đa luồng

- **User threads:** Được hỗ trợ trên kernel và không được kernel quản lý.

- **Kernel threads:** Được hỗ trợ và quản lý trực tiếp bởi hệ điều hành.

### 3.1 Mô hình Many-to-One

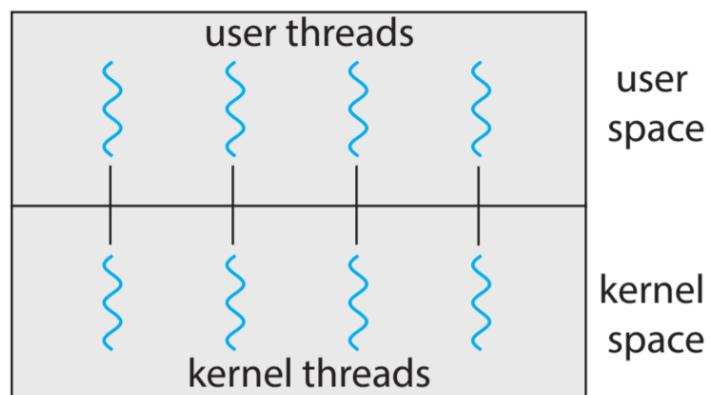
- Nhiều user threads ánh xạ đến một kernel thread
- Khi 1 thread bị khóa, dẫn đến tất cả các thread còn lại bị khóa.
- Vì chỉ có 1 thread thực hiện trên kernel tại một thời điểm nên nhiều thread không thể chạy song song trên hệ xử lý đa lõi.
- Ít hệ thống sử dụng mô hình này.
- Ví dụ: Solaris Green threads, GNU Portable threads



Hình 3.1: Mô hình Many-to-One

### 3.2 Mô hình One-to-One

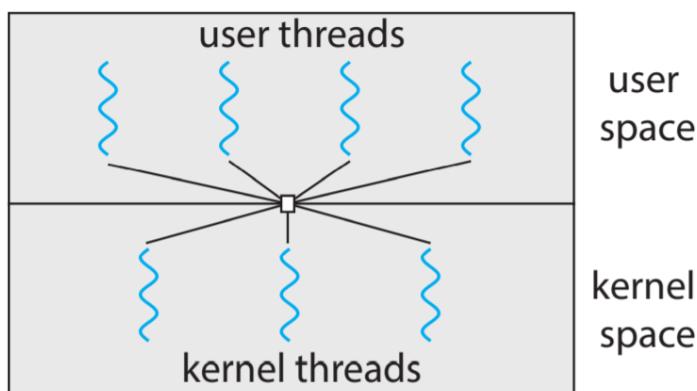
- Mỗi user thread ánh xạ đến một kernel thread.
- Tạo ra user thread cũng tạo ra kernel thread.
- Đồng thời (Concurrency) hơn mô hình many-to-one.
- Số thread trên mỗi process thỉnh thoảng bị giới hạn do chi phí.
- Ví dụ: Windows NT/2000, Linux



Hình 3.2: Mô hình One-to-One

### 3.3 Mô hình Many-to-Many

- Nhiều user thread ánh xạ cho nhiều kernel thread.
- Cho phép hệ điều hành tạo ra một số đủ kernel thread.
- Thực tế ít sử dụng do khó hiện thực.
- Ví dụ: Solaris 2, Windows với ThreadFiber package.



Hình 3.3: Mô hình Many-to-Many

## 4. Các thư viện luồng (Thread Libraries)

- Thread library cung cấp cho lập trình viên API để tạo và quản lý các thread.
- Có 2 cách hiện thực:
  - Thư viện nằm hoàn toàn trong không gian người dùng (user-level).
  - Thư viện cấp độ hệ thống (kernel-level) hỗ trợ bởi OS.

### 4.1 Pthreads

- Là một thư viện hỗ trợ user-level hoặc kernel-level thread.
- Chuẩn POSIX (IEEE 1003.1c) đặc tả API cho các thủ tục tạo thread và đồng bộ thread.
- Phổ biến trong hệ thống UNIX (Linux & Mac OS X).
- Biên dịch và thực thi chương trình multithreaded C trong Linux:
  - \$ gcc source\_file.c -lpthread -o output\_file
  - \$ ./output\_file

### 4.2 Windows Threads

- Để tạo một thread, ta dùng CreateThread(), cũng giống như Pthreads, ta phải truyền những tham số thuộc tính vào lệnh này.

### 4.3 Java Threads

- Java threads được quản lý bởi JVM.
- Thường được hiện thực bằng mô hình thread cung cấp bởi hệ điều hành nền.
- Java threads tạo ra bằng 2 cách:
  - Tạo class mới có nguồn gốc từ Thread class và override hàm run().
  - Hiện thực Runnable Interface.

## 5. Luồng ngầm (Implicit Threading)

- Khi số lượng thread gia tăng, độ chính xác của chương trình sẽ khó hơn với những thread tưởng minh.
- Việc tạo và quản lý thread được thực hiện bởi compilers và run-time libraries hơn là bởi lập trình viên.
- 5 phương pháp để tận dụng xử lý đa lõi thông qua implicit threading:
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks

### 5.1 Thread Pools

- Tạo một số lượng thread nhất định trong một bể điều tiết (pool), các tác vụ đợi được thực thi trên những thread này.
- Lợi ích:
  - Thường nhanh hơn khi thực hiện một yêu cầu trên những thread hiện có so với việc tạo thread mới.
  - Giới hạn được số lượng thread của một chương trình trong kích thước của pool.
  - Cho phép chạy tác vụ với những chiến lược khác nhau
- Trong lập trình Java, `java.util.concurrent` package cung cấp API cho những kiến trúc thread-pool khác nhau.

## 5.2 Fork-Join

- Thread cha sẽ tạo (fork) một hoặc nhiều thread con và đợi thread con kết thúc (terminate) và nhập lại (join) với nhau. Sau đó, thread cha sẽ truy xuất và tổng hợp kết quả trả về của các thread con.
- Một thư viện quản lý số lượng thread được tạo ra và gán các tác vụ lên từng thread. Có thể nói đây là mô hình đồng bộ hóa của Thread Pools.
- Trong Java, fork-join library API được thiết kế để dùng với các thuật toán đệ quy chia để trị (divide-and-conquer) như Quicksort, Mergesort và phải đảm bảo việc thực hiện đồng thời.

## 5.3 OpenMP

- Tập các chỉ thị của compiler cũng như API cho các chương trình C, C++, FORTRAN mà được hỗ trợ lập trình song song trong môi trường chia sẻ bộ nhớ.
- Tạo ra số luồng bằng số lõi của máy.
- Xác định parallel-regions - vùng code cần chạy song song:
  - #pragma omp parallel
  - { /\* code run in parallel \*/ }

## 5.4 Grand Central Dispatch (GCD)

- Công nghệ của Apple cho macOS và iOS.
- Mở rộng cho C, C++, API, và run-time library
- Xác định parallel-sections - vùng code cần chạy song song:
  - ^{/\* code run in parallel \*/}

## 5.5 Intel Threading Building Blocks (TBB)

- Thư viện mẫu cho thiết kế song song chương trình C++.
- Dùng TBB với câu lệnh;
  - parallel for (range body )
  - Ví dụ cho vòng lặp đơn giản:
    - for (int i = 0; i < n; i++) apply(v[i]);
  - parallel for (size t(0), n, [=](size t\_i) {apply(v[i]);});

## 6. Các vấn đề về thread

### 6.1 Ngữ nghĩa của fork() và exec()

- Lệnh fork() nhân đôi chỉ thread gọi nó hay là tất cả các thread?
  - Một số UNIXes có 2 phiên bản fork().
- Lệnh exec() thông thường - thay thế quá trình đang chạy bao gồm tất cả các thread. Nếu vậy, khi gọi exec() sau khi forking, nhân đôi tất cả các thread sẽ không cần thiết, thay vào đó nhân đôi thread đã gọi hợp lý hơn.
- Nhưng nếu không gọi exec() sau khi forking, quá trình nên nhân đôi tất cả các thread.

### 6.2 Xử lý tín hiệu (Signal handling)

- Signal được dùng trong hệ thống UNIX để thông báo cho quá trình rằng sự kiện cụ thể đã xảy ra.
- Signal được xử lý bởi một trong hai loại: mặc định (default) và người dùng định sẵn (user-defined). Và mỗi tín hiệu sẽ thuộc loại mặc định.
- Vậy trong multithreaded, signal sẽ được truyền tới đâu?
  - Truyền tín hiệu đến thread mà tín hiệu được áp dụng.
  - Truyền tín hiệu đến mỗi thread trong process.
  - Truyền tín hiệu đến một thread xác định.
  - Chỉ định một luồng cụ thể để nhận tất cả signal cho một quá trình.

### 6.3 Hủy thread (Thread cancellation)

- Hủy một thread trước khi nó tự kết thúc. Có hai cách:
  - Hủy bất đồng bộ (Asynchronous cancellation): chấm dứt thread ngay lập tức.
  - Hủy được trì hoãn (Deferred cancellation): cho phép thread kiểm tra định kì nếu nó cần bị hủy. Việc hủy được trì hoãn được mặc định sẵn.
- Thực tế lệnh hủy phụ thuộc vào trạng thái của thread. Nếu thread vô hiệu lệnh hủy, lệnh hủy sẽ chờ xử lý tới khi thread đó cho phép.
- Trên hệ thống Linux, lệnh hủy được xử lý thông qua signals.

### 6.4 Lưu trữ cục bộ thread (Thread-local storage TLS)

- Cho phép mỗi thread có phần dữ liệu sao chép riêng.

- Hữu ích khi không có quyền kiểm soát quá trình tạo thread (ví dụ khi dùng thread pool).
- Khác biến cục bộ (local variables), giống với biến static.

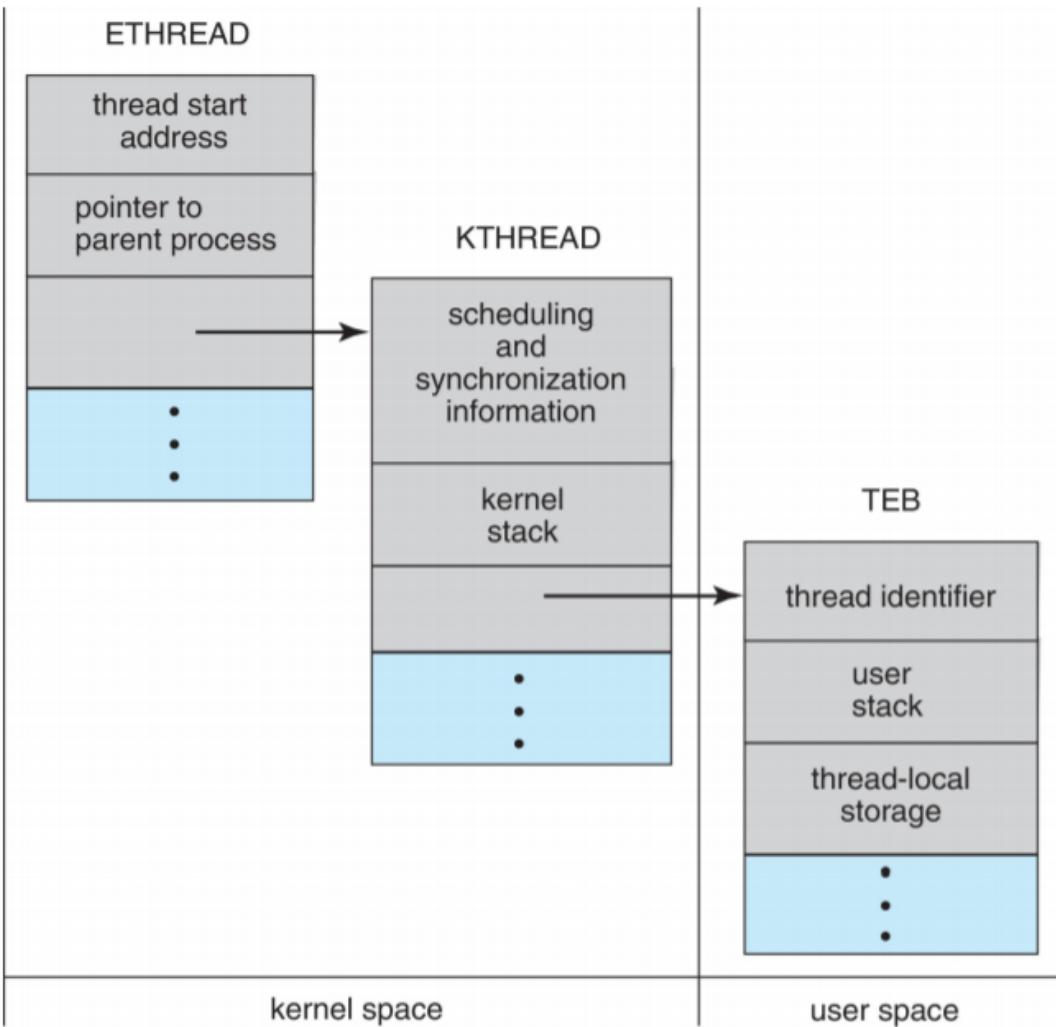
## 6.5 Kích hoạt định thời (Scheduler Activations)

- Mô hình many-to-many và mô hình 2 cấp (user/kernel-level) đều cần giao tiếp để duy trì số lượng kernel thread được phân bổ cho ứng dụng.
- Thường dùng cấu trúc dữ liệu trung gian giữa user và kernel thread - **LightWeight Process (LWP)**.
  - Mỗi LWP được gắn vào kernel thread.
  - Cần tạo ra bao nhiêu LWP?
- **Scheduler activations** cung cấp *upcalls* - cơ chế giao tiếp từ kernel đến trình xử lý upcall trong thread library.
- Cho phép ứng dụng duy trì chính xác số kernel thread.

## 7. Ví dụ của hệ điều hành

### 7.1 Windows Threads

- Windows API - API chính cho các ứng dụng Windows.
- Hiện thực bằng mô hình One-to-One. cấp độ kernel.
- Mỗi thread bao gồm:
  - Thread ID.
  - **Register set**: đại diện trạng thái cho bộ xử lý.
  - **User stack** và **Kernel stack** riêng biệt cho quá trình chạy trong user mode hay kernel mode.
  - Vùng lưu trữ dữ liệu riêng (**Private data storage area**) được dùng bởi run-time libraries và dynamic link libraries (DLLs).
- Cấu trúc dữ liệu chính của thread bao gồm:
  - **ETHREAD** (executive thread block) : thuộc không gian kernel.
  - **KTHREAD** (kernel thread block) : thuộc không gian kernel.
  - **TEB** (thread environment block): thuộc không gian người dùng.



Hình 7.1: Mô tả cấu trúc dữ liệu của Windows thread

## 7.2 Linux Threads

- Linux không phân biệt giữa quá trình (process) và luồng (thread), thông thường được gọi chung là tác vụ (task).
- Việc tạo thread được tạo bởi lệnh hệ thống: `clone()`.
- Lệnh `clone()` cho phép tác vụ con chia sẻ không gian địa chỉ của tác vụ cha (process).

## 8. Câu hỏi

1. Các thread trong quá trình xử lý đa luồng (multithreaded) chia sẻ với nhau về:
  - a. Vùng nhớ stack
  - b. **Biến toàn cục**
  - c. Vùng nhớ heap
  - d. Program counter
2. Nhận định **sai** về thực thi đồng thời và thực thi song song:
  - a. Có thể có thực thi đồng thời nhưng không song song.
  - b. Có thể có đa nhiệm trên một bộ xử lý đơn lõi.
  - c. **Hai nhiệm vụ T1 và T2 là đồng thời thì T1 hoàn thành cùng lúc với T2**
  - d. Xử lý song song là khi các nhiệm vụ chạy cùng một lúc
3. Dùng định luật Amdahl, độ tăng tốc cho ứng dụng có 70% xử lý song song và 8 lõi xử lý là:
  - a. **2,58**
  - b. 1,36
  - c. 1,27
  - d. 2,96
4. Xác định vấn đề nào sau đây cần xử lý nhiệm vụ song song:
  - a. Xử lý đa luồng cho chương trình sắp xếp (sorting).
  - b. **Xử lý đa luồng cho chương trình kiểm tra bảng Sudoku.**
  - c. Xử lý đa luồng cho chương trình tính tổng hai ma trận.
  - d. Xử lý đa luồng cho chương trình thống kê.

5. Xem đoạn chương trình sau trong linux và cho biết có bao nhiêu luồng được tạo ra, nếu tính cả luồng chính:

```
pid_t pid;  
pid = fork();  
if (pid == 0) {  
    fork();  
}  
else fork();
```

a. 3

b. 4

c. 5

d. 6

6. Nhận định nào sau đây đúng về mô hình xử lý đa luồng:

a. Mô hình One-to-One đồng thời hơn Many-to-One.

b. Mô hình Many-to-One ít được sử dụng do không thể chạy song song trên hệ xử lý đa lõi.

c. Mô hình Many-to-Many ít sử dụng do khó hiện thực.

d. Tất cả đều đúng.

7. Các thuật toán đệ quy chia để trị (divide-and-conquer) như Quicksort, Mergesort dùng phương pháp nào để tận dụng xử lý đa lõi thông qua implicit threading:

a. OpenMP

b. Fork-Join

c. Thread Pools

d. Grand Central Dispatch (GCD)

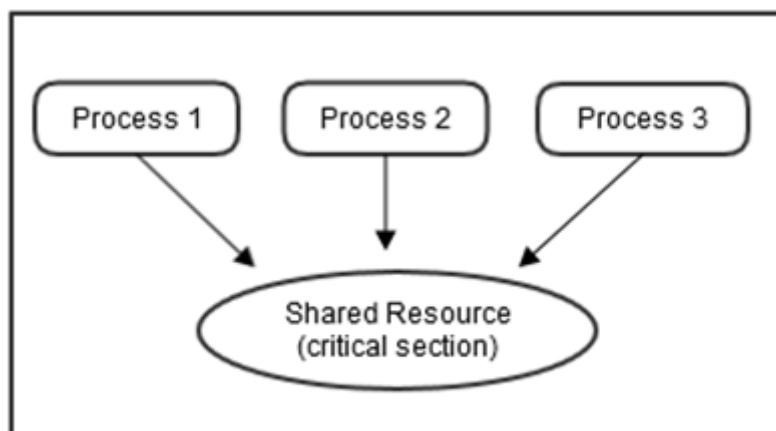
## CHƯƠNG 5: ĐỒNG BỘ VÀ GIẢI QUYẾT TRANH CHẤP

### 1. Đặt vấn đề

Khảo sát các process/thread thực thi đồng thời và chia sẻ dữ liệu (qua shared memory, file). Nếu không có sự kiểm soát khi truy cập các dữ liệu chia sẻ thì có thể đưa đến ra trường hợp không nhất quán dữ liệu (data inconsistency).

Truy xuất bộ nhớ chung là 1 trong nhiều hoạt động tương tranh giữa các process. Vấn đề tương tranh trên 1 tài nguyên dùng chung là vấn đề lớn cần phải giải quyết triệt để vì nếu nhiều process truy xuất đồng thời vào 1 tài nguyên dùng chung mà không có sự kiểm soát thì dễ xảy ra lỗi làm hư hỏng tài nguyên (điều kiện Race).

Để duy trì sự nhất quán dữ liệu, hệ thống cần có cơ chế bảo đảm sự thực thi có trật tự lần lượt thao tác lên dữ liệu chia sẻ của các process đồng thời. Do đó, cần có cơ chế đồng bộ hoạt động của các process này.



### 1.1. Race Conditions

Race condition (Tình huống tương tranh) là một tình huống xảy ra khi nhiều threads cùng truy cập và cùng lúc muốn thay đổi dữ liệu (có thể là một biến, một row trong database, một vùng shared data, memory , etc...). Vì thuật toán chuyển đổi việc thực thi giữa các threads có thể xảy ra bất cứ lúc nào, nên không thể biết được thứ tự của các threads truy cập và thay đổi dữ liệu đó sẽ dẫn đến giá trị của data sẽ không như mong muốn. Kết quả sẽ phụ thuộc vào thuật toán thread scheduling thứ tự thực thi các thread của hệ điều hành các lệnh thao tác dữ liệu. Quá trình các threads thực thi lệnh trông như 1 cuộc đua giữa các vận động viên điền kinh olympic vì vậy có thể liên tưởng đến thuật ngữ "Race condition".

### 1.2. Critical sections

Không phải tất cả các đoạn code đều cần được giải quyết vấn đề race condition mà chỉ những đoạn code có chứa các thao tác lên dữ liệu chia sẻ. Đoạn code này được gọi là vùng tranh chấp (critical section, CS). Là đoạn mã mà khi khởi chạy chúng trên nhiều thread sẽ dẫn tới việc đọc ghi chung biến, file ... (Tổng quát hơn là dữ liệu).

Lưu ý: việc khởi chạy nhiều hơn một thread bên trong cùng một ứng dụng không tự phát sinh vấn đề.

### 1.3. Race Conditions trong Critical Sections

Khi multiple threads thực thi critical section, race conditions sẽ xảy ra. Cụ thể hơn, khi 2 hay nhiều threads cùng sử dụng chung một resource, khi mà thứ tự thao tác với resource có ý nghĩa quan trọng được gọi là race conditions. Đoạn code dẫn tới race conditions được gọi là critical sections.

## 2. Vấn đề vùng tranh chấp (CriticalSection Problem)

Là vấn đề về việc tìm một cách thiết kế một giao thức (một cách thức) nào đó để các process có thể phối hợp với nhau hoàn thành nhiệm vụ của nó. phải bảo đảm sự loại trừ tương hỗ (Mutual Exclusion, mutex), tức là khi **một process đang thực thi trong vùng tranh chấp, không có process nào khác đồng thời thực thi các lệnh trong vùng tranh chấp.**

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

Một lời giải cho vấn đề vùng tranh chấp (CS Problem) phải đảm bảo được 3 tính chất sau :

- **Loại trừ tương hỗ (Mutual Exclusion)** : Khi một process P đang thực thi trong **vùng tranh chấp (CS)** của nó thì **không có process Q nào khác đang thực thi trong CS của Q.**
- **Phát triển (Progress)** : Một tiến trình tạm dừng bên ngoài CS không được ngăn cản các tiến trình khác vào CS. Nếu không có quá trình nào đang thực thi trong **vùng tranh chấp (CS)** và có ít nhất một quá trình muốn vào **vùng tranh chấp**, thì **chỉ có những quá trình đang không thực thi trong **vùng remainder (RS)** mới có quyền quyết định lựa chọn quá trình kế tiếp vào **vùng tranh chấp** và quyết định đó không được phép trì hoãn vô hạn định**
- **Chờ đợi giới hạn (Bounded Waiting)** : Khi một quá trình muốn vào **vùng tranh chấp (CS)**, thì **từ khi yêu cầu đến khi được đáp ứng là khoảng thời gian có hạn (bounded or limit).** Không được xảy ra tình trạng đói tài nguyên (starvation).

### 3. Các bài toán đồng bộ

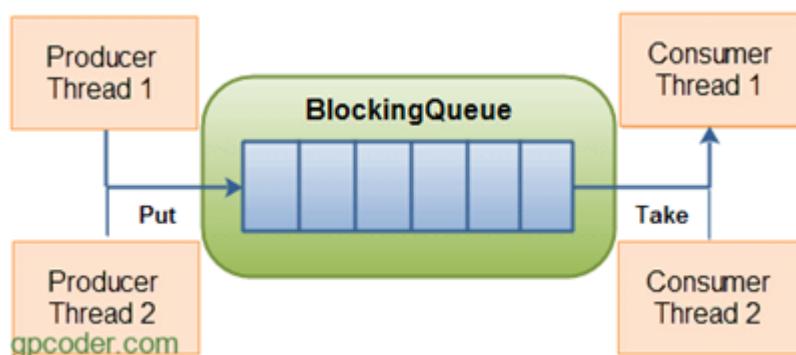
#### 3.1. Bài toán Producer/Consumer ( Bounded buffer)

##### 3.1.1. Mô tả

Vấn đề mô tả hai đối tượng nhà sản xuất (Producer) và người tiêu dùng (Consumer), cả hai cùng chia sẻ một bộ đệm có kích thước cố định được sử dụng như một hàng đợi (queue).

- **Producer:** công việc của nhà sản xuất là tạo dữ liệu, đưa nó vào bộ đệm và bắt đầu lại.
- **Consumer:** công việc người tiêu dùng là tiêu thụ dữ liệu (nghĩa là loại bỏ nó khỏi bộ đệm), từng phần một và xử lý nó. Consumer và Producer hoạt động song song với nhau.

Vấn đề là đảm bảo rằng nhà sản xuất không thể thêm dữ liệu vào bộ đệm nếu nó đầy và người tiêu dùng không thể xóa dữ liệu khỏi bộ đệm trống, đồng thời đảm bảo an toàn cho luồng (thread-safe).



##### 3.1.2. Giải pháp

Ý tưởng:

Giải pháp cho nhà sản xuất là đi ngủ (wait) nếu bộ đệm đầy. Lần tiếp theo người tiêu dùng xóa một mục khỏi bộ đệm, nó đánh thức (notify) nhà sản xuất, bắt đầu đưa dữ liệu vào bộ đệm.

Theo cách tương tự, người tiêu dùng có thể đi ngủ (wait) nếu thấy bộ đệm trống. Lần tiếp theo nhà sản xuất đưa dữ liệu vào bộ đệm, nó đánh thức (notify) người tiêu dùng đang ngủ (wait).

Trong khi làm tất cả điều này, phải đảm bảo an toàn cho luồng (thread safe).

```
do {  
    . . .  
    // produce an item in nextp  
    . . .  
    wait(empty);  
    wait(mutex);  
    . . .  
    // add nextp to buffer  
    . . .  
    signal(mutex);  
    signal(full);  
}while (TRUE);
```

Figure 5.9 The structure of the producer process.

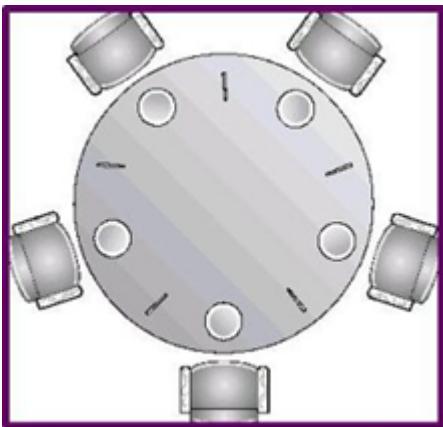
```
do {  
    wait(full);  
    wait(mutex);  
    . . .  
    // remove an item from buffer to nextc  
    . . .  
    signal(mutex);  
    signal(empty);  
    . . .  
    // consume the item in nextc  
}while (TRUE);
```

Figure 5.10 The structure of the consumer process.

### 3.2. Bài toán "Bữa tối của các triết gia" ("Dining Philosophers")

#### 3.2.1. Mô tả

Có năm nhà triết gia, vừa suy nghĩ vừa ăn tối. Các triết gia ngồi trên một bàn tròn, trước mặt họ là các đĩa thức ăn, mỗi người một đĩa. Có 5 chiếc đũa được đặt xen kẽ giữa các triết gia.



Bài toán được phát biểu như sau: "Khi một triết gia suy nghĩ, ông ta không giao tiếp với các triết gia khác. Thỉnh thoảng, một triết gia cảm thấy đói và cố gắng chọn hai chiếc đũa gần nhất (hai chiếc đũa nằm giữa ông ta với hai lảng giềng trái và phải). Một triết gia có thể lấy chỉ một chiếc đũa tại một thời điểm. Chú ý, ông ta không thể lấy chiếc đũa mà nó đang được dùng bởi người láng giềng. Khi một triết gia đói và có hai chiếc đũa cùng một lúc, ông ta ăn mà không đặt đũa xuống. Khi triết gia ăn xong, ông ta đặt đũa xuống và bắt đầu suy nghĩ tiếp."

Tìm cách để không ai phải chết đói là vấn đề của bài toán.

Bài toán này minh họa sự khó khăn trong quá trình phân phối cấp phát nhiều tài nguyên giữa các tiến trình mà không bị khoá chết (deadlock) và đói tài nguyên (starvation).

### 3.2.2. Giải pháp

Dữ liệu chia sẻ: semaphore chopstick[5];

Khởi đầu các biến đều là 1;

Giải pháp trên có thể gây ra deadlock – Khi tất cả triết gia đói bụng cùng lúc và đồng thời cầm chiếc đũa bên tay trái deadlock.

Triết gia thứ  $i$ :

```
do {  
    wait(chopstick [ i ]);  
    wait(chopstick [ (i + 1) % 5 ]);  
    ...  
    eat  
    ...  
    signal(chopstick [ i ]);  
    signal(chopstick [ (i + 1) % 5 ]);  
    ...  
    think  
    ...  
} while (1);
```

Semaphore là lời giải kinh điển cho bài toán bữa tối của các triết gia (dining philosophers), mặc dù nó không ngăn được hết các Deadlock. Để giải quyết Deadlock, ở vị trí cuối cùng ta sẽ cho triết gia đó được lấy đũa bên phải mình trước sau đó mới lấy đũa bên trái trong khi các triết gia khác sẽ lấy đũa bên trái trước. Điều đó chắc chắn rằng triết gia vị trí cuối luôn sẵn sàng ăn đầu tiên khi mà các triết gia khác đang cố lấy chiếc đũa bên phải của mình. kết quả là, không ai phải chết đói.

Một số giải pháp khác giải quyết được deadlock

- Cho phép nhiều nhất 4 triết gia ngồi vào cùng một lúc
- Cho phép triết gia cầm các đũa chỉ khi cả hai chiếc đũa đều sẵn sàng (nghĩa là tác vụ cầm các đũa phải xảy ra trong CS)
- Triết gia ngồi ở vị trí lẻ cầm đũa bên trái trước, sau đó mới đến đũa bên phải, trong khi đó triết gia ở vị trí chẵn cầm đũa bên phải trước, sau đó mới đến đũa bên trái

## 3.3. Bài toán Readers-Writers

### 3.3.1. Mô tả

Bộ đọc-bộ ghi (Readers-Writers) là một đối tượng dữ liệu (như một tập tin hay mẩu tin) được chia sẻ giữa nhiều quá trình đồng hành. Một số trong các quá trình có thể chỉ cần đọc nội dung của đối tượng được chia sẻ, ngược lại một vài quá trình khác

cần cập nhật (nghĩa là đọc và ghi) trên đối tượng được chia sẻ. Chúng ta phân biệt sự khác nhau giữa hai loại quá trình này bằng cách gọi các quá trình chỉ đọc là bộ đọc và các quá trình cần cập nhật là bộ ghi. Chú ý, nếu hai bộ đọc truy xuất đối tượng được chia sẻ cùng một lúc sẽ không có ảnh hưởng gì. Tuy nhiên, nếu một bộ ghi và vài quá trình khác (có thể là bộ đọc hay bộ ghi) truy xuất cùng một lúc có thể dẫn đến sự hỗn độn.

Để đảm bảo những khó khăn này không phát sinh, chúng ta yêu cầu các bộ ghi có truy xuất loại trừ lẫn nhau tới đối tượng chia sẻ. Việc đồng bộ hóa này được gọi là bài toán bộ đọc-bộ ghi. Bài toán bộ đọc-bộ ghi có một số biến dạng liên quan đến độ ưu tiên. Dạng đơn giản nhất là bài toán bộ đọc trước-bộ ghi (first reader-writer). Trong dạng này yêu cầu không có bộ đọc nào phải chờ ngoại trừ có một bộ ghi đã được cấp quyền sử dụng đối tượng chia sẻ. Nói cách khác, không có bộ đọc nào phải chờ các bộ đọc khác để hoàn thành đơn giản vì một bộ ghi đang chờ. Bài toán bộ đọc sau-bộ ghi (second readers-writers) yêu cầu một khi bộ ghi đang sẵn sàng, bộ ghi đó thực hiện việc ghi của nó sớm nhất có thể. Nói một cách khác, nếu bộ ghi đang chờ truy xuất đối tượng, không có bộ đọc nào có thể bắt đầu việc đọc.

- Có một database hoặc file, nhiều Readers (để đọc) và nhiều Writers (để ghi) dữ liệu vào database.
- Khi một Writer đang truy cập database/file thì không một quá trình nào khác được truy cập.
- Nhiều Readers có thể cùng lúc đọc database/file

### 3.3.2. Giải pháp

Dữ liệu chia sẻ: semaphore mutex = 1;  
semaphore rt\_mutex = 1;  
int read\_count = 0;

```

do {
    wait(rw_mutex);
    . . .
    /* writing is performed */
    . . .
    signal(rw_mutex);
} while (true);

```

The structure of a writer process.

```

do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    . . .
    /* reading is performed */
    . . .
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);

```

The structure of a reader process.

- mutex: "bảo vệ" biến readcount
- wrt
  - Bảo đảm mutual exclusion đối với các writer
  - Được sử dụng bởi reader đầu tiên hoặc cuối cùng vào hay ra khỏi vùng tranh chấp.
- Nếu một writer đang ở trong CS và có n reader đang đợi thì một reader được xếp trong hàng đợi của wrt và n – 1 reader kia trong hàng đợi của mutex
- Khi writer thực thi signal(wrt), hệ thống có thể phục hồi thực thi của một trong các reader đang đợi hoặc writer đang đợi.

#### 4. Phân loại giải pháp cho loại trừ tương hỗ

Có 2 nhóm giải pháp chính :

★ Nhóm giải pháp Busy Waiting :

Tính chất :

- Tiếp tục tiêu thụ CPU trong khi chờ đợi vào miền găng ( thông qua việc kiểm tra điều kiện vào CS liên tục).
- Không đòi hỏi sự trợ giúp của hệ điều hành.

**Cơ chế chung :**

**While (chưa có quyền) do nothing() ;**

CS;

**Từ bỏ quyền sử dụng CS**

Cơ chế chung của nhóm giải pháp Busy Waiting.

**Bao gồm một vài loại :**

- Sử dụng các biến cờ hiệu.
- Sử dụng việc kiểm tra luân phiên.
- Giải pháp của Peterson.
- Cấm ngắt (giải pháp phần cứng – hardware).
- Chỉ thị TSL (giải pháp phần cứng – hardware).

★ **Nhóm giải pháp Sleep & Wakeup:**

**Tính chất :**

- Từ bỏ CPU khi chưa được vào CS.
- Cần sự hỗ trợ từ hệ điều hành (để đánh thức process và đưa process vào trạng thái blocked).

**Cơ chế chung :**

**if (chưa có quyền) Sleep() ;**

CS;

**Wakeup (somebody);**

Cơ chế chung của nhóm giải pháp Sleep & Wakeup.

**Bao gồm một vài loại :**

- Semaphore.
- Monitor.

- Message.

## 5. Giải thuật Peterson

### 5.1. Giải thuật Peterson cho 2 process

Là một lời giải trọn vẹn thỏa mãn tất cả điều kiện của bài toán tương hỗ cho 2 process.

Giải thuật Peterson yêu cầu hai biến chia sẻ chung cho cả hai process là:

```
int turn;
boolean flags[2];
Process Pi , với i = 0 hay 1
```

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}
```

Figure 5.5 The definition of the compare\_and\_swap() instruction.

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the compare\_and\_swap() instruction.

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
```

critical section

```
flag[i] = FALSE;
```

remainder section

```
} while (TRUE);
```

## 5.2. Tính đúng đắn

Mutual exclusion được đảm bảo :

- Chứng minh bằng phản chứng: Nếu P0 và P1 cùng ở trong CS thì  $\text{flag}[0] = \text{flag}[1] = \text{true}$ , suy ra từ điều kiện của vòng lặp while sẽ có  $\text{turn} = 0$  (trong P0) và  $\text{turn} = 1$  (trong P1). Điều không thể xảy ra.

Thỏa yêu cầu về progress và bounded waiting :

- Pi không thể vào CS nếu và chỉ nếu bị kẹt tại vòng lặp while() với điều kiện  $\text{flag}[j] = \text{true}$  và  $\text{turn} = j$ .
- Nếu Pj không muốn vào CS thì  $\text{flag}[j] = \text{false}$  và do đó Pi có thể vào CS.
- Nếu Pj đã bật  $\text{flag}[j] = \text{true}$  và đang chờ tại while() thì có chỉ hai trường hợp là  $\text{turn} = i$  hoặc  $\text{turn} = j$ .
- Nếu  $\text{turn} = i$  thì Pi vào CS. Nếu  $\text{turn} = j$  thì Pj vào CS nhưng sẽ bật  $\text{flag}[j] = \text{false}$  khi thoát ra cho phép Pi vào CS.
- Nhưng nếu Pj có đủ thời gian bật  $\text{flag}[j] = \text{true}$  thì Pj cũng phải gán  $\text{turn} = i$
- Vì Pi không thay đổi trị của biến turn khi đang kẹt trong vòng lặp while(), Pi sẽ chờ để vào CS nhiều nhất là sau một lần Pj vào CS (bounded waiting).

## 5.3. Đánh giá

Khuyết điểm của các giải pháp software

- Các process khi yêu cầu được vào vùng tranh chấp đều phải liên tục kiểm tra điều kiện (busy waiting), tốn nhiều thời gian xử lý của CPU
- Nếu thời gian xử lý trong vùng tranh chấp lớn, một giải pháp hiệu quả nên có cơ chế block các process cần đợi.

Các giải pháp phần cứng (hardware)

- Cấm ngắt (disable interrupts)
- Dùng các lệnh đặc biệt

## 6. Lệnh TestAndSet

Đây là một giải pháp đòi hỏi sự trợ giúp của cơ chế phần cứng. Nhiều máy tính cung cấp một chỉ thị đặc biệt cho phép kiểm tra và cập nhật nội dung một vùng nhớ trong một thao tác không thể phân chia, gọi là chỉ thị Test-and-Set Lock (TSL), đọc và ghi một biến trong một thao tác atomic (không chia cắt được).

```
boolean TestAndSet(boolean *target) {  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

Figure 5.3 The definition of the TestAndSet() instruction.

---

```
do {  
    while (TestAndSetLock(&lock))  
        ; // do nothing  
  
    // critical section  
    lock = FALSE;  
  
    // remainder section  
}while (TRUE);
```

Figure 5.4 Mutual-exclusion implementation with TestAndSet().

Nếu có hai chỉ thị TSL xử lý đồng thời (trên hai bộ xử lý khác nhau), chúng sẽ được xử lý tuần tự. Có thể cài đặt giải pháp truy xuất độc quyền với TSL bằng cách sử dụng thêm một biến lock, được khởi gán là FALSE. Tiến trình phải kiểm tra giá trị của biến lock trước khi vào vùng tranh chấp, nếu lock = FALSE, tiến trình có thể vào vùng tranh chấp.

Mutual exclusion được bảo đảm: nếu Pi vào CS, các process Pj khác đều đang busy waiting

- Khi Pi ra khỏi CS, quá trình chọn lựa process Pj vào CS kế tiếp là tùy ý không bảo đảm điều kiện bounded waiting. Do đó có thể xảy ra starvation (bị bỏ đói)

- Các processor (ví dụ Pentium) thông thường cung cấp một lệnh đơn là Swap(a, b) có tác dụng hoán chuyển nội dung của a và b.
  - Swap(a, b) cũng có ưu nhược điểm như TestAndSet

Biến chia sẻ lock được khởi tạo giá trị false

- Mỗi process Pi có biến cục bộ key
- Process Pi nào thấy giá trị lock = false thì được vào CS.

Process Pi sẽ loại trừ các process Pj khác khi thiết lập lock = true

Biến chia sẻ (khởi tạo là false): bool lock;

```
bool waiting [n];
```

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}

do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        /* do nothing */
    /* critical section */
    lock = 0;
    /* remainder section */
} while (true);

do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
    // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
    // remainder section
}while (TRUE);
```

Figure 5.5 The definition of the compare\_and\_swap() instruction.

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        /* do nothing */
    /* critical section */
    lock = 0;
    /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the compare\_and\_swap() instruction.

Không thỏa mãn bounded waiting

Giải thuật dùng TestAndSet thỏa mãn yêu cầu:

- Cấu trúc dữ liệu dùng chung (khởi tạo là false) bool waiting[ n ]; bool lock;

- Mutual exclusion: Pi chỉ có thể vào CS nếu và chỉ nếu hoặc waiting[ i ] = false, hoặc key = false
  - key = false chỉ khi TestAndSet (hay Swap) được thực thi
  - Process đầu tiên thực thi TestAndSet mới có key == false; các process khác đều phải đợi • waiting[ i ] = false chỉ khi process khác rời khỏi CS
  - Chỉ có một waiting[ i ] có giá trị false
- Progress: chứng minh tương tự như mutual exclusion
- Bounded waiting: waiting in the cyclic order

## 7. Mutex lock

Khóa hoặc mutex (từ loại trừ lẫn nhau ) là một cơ chế đồng bộ hóa để thực thi các giới hạn truy cập vào tài nguyên trong môi trường có nhiều luồng thực thi . Một khóa được thiết kế để thực thi chính sách kiểm soát đồng thời loại trừ lẫn nhau .

- Cũng giống như với khóa phần cứng, bước thu sẽ chặn quy trình nếu khóa được sử dụng bởi quy trình khác và cả hoạt động thu và giải phóng đều là atomic.
- Một vấn đề với việc triển khai là vòng lặp bận được sử dụng để chặn các quy trình trong giai đoạn thu nhận. Các loại khóa này được gọi là spinlocks , bởi vì CPU chỉ ngồi và quay trong khi chặn quá trình.
- Spinlocks gây lãng phí cho chu kỳ cpu và là một ý tưởng thực sự tồi tệ đối với các máy đơn luồng cpu đơn, vì spinlock chặn toàn bộ máy tính và không cho phép bất kỳ quá trình nào khác giải phóng khóa. (Cho đến khi bộ lập lịch khởi động quy trình quay vòng của cpu.)
- Mặt khác, spinlocks không phát sinh chi phí chuyển đổi ngữ cảnh, vì vậy chúng được sử dụng hiệu quả trên các máy đa luồng khi dự kiến khóa sẽ được phát hành sau một thời gian ngắn.

## 8. Semaphore

Semaphore là một cấu trúc dữ liệu, được dùng để đồng bộ tài nguyên và đồng bộ hoạt động.

Khi được dùng với mục đích đồng bộ tài nguyên, semaphore tương tự như một bộ các chìa khóa dự phòng. Nếu một thread lấy được một chiếc chìa khóa, thread đó được phép truy cập vào tài nguyên. Nhưng nếu không còn chiếc chìa khóa nào, thread đó phải đợi cho tới khi một thread khác trả lại chìa khóa dự phòng. Nhờ vậy, race condition sẽ bị ngăn chặn.

### Semaphore Structure:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

### Wait Operation:

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

### Signal Operation:

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Mỗi Semaphore có một giá trị nguyên của nó và một danh sách các process. Khi các process chưa sẵn sàng để thực thi thì sẽ được đưa vào danh sách này. Danh sách này còn gọi là hàng đợi semaphore. Lưu ý: Các process khi ra khỏi hàng đợi semaphore sẽ vào hàng đợi Ready để chờ lấy CPU thực thi.

- Khi S.value  $\geq 0$ : value chính là số process có thể thực thi wait(S) mà không bị blocked
- Khi S.value < 0: trị tuyệt đối của value chính là số process đang đợi trên hàng đợi semaphore

Việc hiện thực Semaphore phải đảm bảo tính chất Atomic và mutual exclusion: tức không được xảy ra trường hợp 2 process cùng đang ở trong thân lệnh wait(S) và signal(S) (cùng semaphore S) tại một thời điểm (ngay cả với hệ thống multiprocessor)  $\Rightarrow$  do đó, đoạn mã định nghĩa các lệnh wait(S) và signal(S) cũng chính là vùng tranh chấp Giải pháp cho vùng tranh chấp wait(S) và signal(S): Uniprocessor: có thể dùng cơ chế cấm ngắt (disable interrupt). Nhưng phương pháp này không hiệu quả trên hệ thống multiprocessor. Multiprocessor: có thể dùng các giải pháp software (như giải Peterson và Bakery) hoặc giải pháp hardware (TestAndSet, Swap). Vùng tranh chấp của các tác vụ wait(S) và signal(S) thông thường rất nhỏ: khoảng 10 lệnh. Vì CS rất nhỏ nên chi phí cho busy waiting sẽ rất thấp.

## 9. Monitor

Monitor là đối tượng được cung cấp sẵn bởi hệ thống. Về cấu trúc, monitor cũng gồm nhiều thuộc tính dữ liệu và nhiều tác vụ chức năng

Sự khác biệt giữa Monitor và 1 đối tượng (module phần mềm) bình thường được thể hiện bởi 2 tính chất sau : 1. trong monitor, ta có thể định nghĩa nhiều biến điều kiện. Biến điều kiện chỉ có tên, không có kiểu và không có giá trị nên ta không dùng nó như biến bình thường. Chỉ có 2 tác vụ định sẵn có thể truy xuất biến điều kiện : - hàm wait(cond) : bắt process ngủ chờ trên biến điều kiện cond. - hàm signal(cond) : đánh thức các process đang ngủ chờ trên biến cond. 2. Về mặt điều khiển, Monitor chỉ cho phép tối đa 1 process được vào thi hành 1 tác vụ nào đó của Monitor tại từng thời điểm. Nhờ tính chất này, ta dễ dàng giải quyết việc loại trừ tương hỗ giữa các process khi chúng truy xuất đồng thời 1 tài nguyên dùng chung nào đó bằng cách đặt mỗi đoạn CS vào 1 tác vụ riêng và đặt tất cả các tác vụ này trong một Monitor nào đó

## 10. Deadlock

Deadlock là một trạng thái mà ở đó một tập các tiến trình bị chặn vì mỗi tiến trình đang chiếm giữ tài nguyên và chờ đợi được cấp phát tài nguyên khác được giữ bởi tiến trình khác.

Nói cách khác, mỗi tiến trình trong hệ thống đang chờ để được cấp phát tài nguyên đang bị chiếm giữ bởi tài nguyên khác. Nếu tài nguyên đó không được giải phóng để tiến trình khác có thể sử dụng, thì tiến trình đang chờ lấy tài nguyên sẽ chờ mãi, chờ mãi dẫn tới Khó chết ( Deadlock )

### Các điều kiện phát sinh Deadlock

- Loại trừ tương hỗ (Mutual Exclusion): Tại một thời điểm, tài nguyên không thể chia sẻ được hệ thống cấp phát cho một tiến trình duy nhất. Tiến trình khác không thể sử dụng cho đến khi tài nguyên được giải phóng.
- Giữ và chờ (Hold and Wait) : Mỗi tiến trình trong tập hợp tiến trình đang giữ một tài nguyên và chờ đợi để được cấp phát một tài nguyên mới.
- Không có quyền ưu tiên (No Preemption): Một tiến trình không thể chiếm giữ tài nguyên cho đến khi tài nguyên đó được giải phóng bởi tiến trình đang sử dụng nó.
- Tồn tại chu kỳ chờ (Circular wait): Các tiến trình giữ một tài nguyên và chờ nhận một tài nguyên khác bởi tiến trình khác. Chúng nối đuôi nhau tạo thành vòng tròn. Chờ vô tận.

Deadlock chỉ xảy ra khi có đủ 4 điều kiện trên.

## 11. Câu hỏi trắc nghiệm

Câu 1. Khi một process chuẩn bị vào hay ra khỏi một vùng Critical Section thì phải:

- a) Xin phép hệ điều hành

b) Phất cờ hiệu khi vào và trả khi ra

c) Cả hai ý trên

Câu 2. Một semaphore là một biến số nguyên được chia sẻ \_\_\_\_\_

a) không thể giảm xuống dưới 0

b) không thể nhiều hơn 0

c) không thể giảm xuống dưới một

d) không thể nhiều hơn một

Câu 3. Đồng bộ hóa quy trình có thể được thực hiện trên \_\_\_\_\_

a) cấp độ phần cứng

b) cấp độ phần mềm

c) cả cấp độ phần cứng và phần mềm

d) không có

Câu 4. Spinlocks là gì?

a) Chu trình CPU lãng phí khóa trên các phần quan trọng của chương trình

b) Khóa tránh lãng phí thời gian trong các công tắc ngữ cảnh

c) Khóa hoạt động tốt hơn trên các hệ thống đa bộ xử lý

d) Tất cả các câu

Câu 5. Nếu giá trị semaphore là âm \_\_\_\_\_

a) cường độ của nó là số lượng quá trình đang chờ trên semaphore đó

b) thì không hợp lệ

c) không có thao tác nào có thể được thực hiện thêm cho đến khi hoạt động tín hiệu được thực hiện trên nó đề cập

# CHƯƠNG 6: CÁC VÍ DỤ VỀ ĐỒNG BỘ

## 1 Các vấn đề kinh điển của đồng bộ

### 1.1. The bounded-buffer problem (Producer-consumer problem)

-Vấn đề này thường được sử dụng để minh họa cho sức mạnh của các chiến thuật đồng bộ hóa.

- Giả sử rằng có n buffer, mỗi buffer có thể chứa được 1 item. *empty* và *full* semaphore đếm số buffer trống và buffer đầy. Ban đầu, *empty semaphore* = n và *full semaphore* = 0.

### 1.2. The Readers-Writers Problem

-Cho 1 cơ sở dữ liệu được chia sẻ giữa các tiến trình đồng thời. Trong các tiến trình đồng thời này, có 1 vài tiến trình chỉ muốn đọc cơ sở dữ liệu, trong khi 1 vài tiến trình khác lại muốn cập nhật (cả đọc và viết) cơ sở dữ liệu. Chúng ta phân biệt giữa 2 loại tiến trình này bằng cách đề cập đến loại tiến trình đầu bằng thuật ngữ readers và loại tiến trình còn lại là writers. Rõ ràng là, nếu một writer và các tiến trình khác (có thể là writer hoặc reader) truy cập cơ sở dữ liệu đồng thời, vấn đề hỗn loạn sẽ xảy ra.

-Để đảm bảo rằng những vấn đề này không xuất hiện, chúng ta cần các writer có sự độc chiếm truy cập tới cơ sở dữ liệu được chia sẻ khi đang thực hiện ghi đến cơ sở dữ liệu đó. Vấn đề đồng bộ này được gọi là “readers - writers problem”. Bởi vì đây là một vấn đề kinh điển được chỉ ra từ sớm, nó luôn được dùng để kiểm tra mọi chiến thuật đồng bộ hóa mới.

-"Readers-writers problem" có nhiều biến thể, tất cả biến thể này đều bao gồm các ưu tiên. Ưu tiên đơn giản nhất, cũng được coi là vấn đề readers- writers đầu tiên, yêu cầu rằng không có reader nào phải đợi trừ khi 1 writer đã có được quyền để sử dụng đối tượng chia sẻ. Nói cách khác, không có reader nào nên đợi những reader khác hoàn thành đơn giản bởi 1 writer khác đang đợi. Những readers-writers thứ hai yêu cầu rằng, nếu 1 writer đang sẵn sàng, thì writer đó thực hiện việc ghi càng sớm càng tốt. Nói cách khác, nếu 1 writer đang đợi để được truy cập vào đối tượng thì không có reader nào được bắt đầu đọc.

- Giải pháp cho vấn đề này có thể gây ra hiện tượng starvation(chết đói). Trong trường hợp đầu tiên, writers có thể bị đói; trường hợp thứ hai thì reader có thể bị đói. Vì lý do này mà các biến thể của vấn đề được đưa ra. Tiếp theo, chúng ta sẽ đưa ra một giải pháp cho vấn đề readers-writers đầu tiên.

-Trong giải pháp về vấn đề readers-writers đầu tiên, những tiến trình đọc sẽ chia sẻ vùng cấu trúc dữ liệu sau:

```
semaphore rw_mutex = 1;
```

```
semaphore mutex = 1;
```

```
int read_count = 0;
```

-Semaphore nhị phân rw\_mutex và mutex được khởi tạo với giá trị bằng 1, read\_count là 1 semaphore đếm với giá trị khởi tạo bằng 0. Semaphore rw\_mutex được sử dụng chung cho cả reader và writer. Semaphore mutex được sử dụng để đảm bảo sự loại trừ tương hỗ khi mà biến read\_count được cập nhật. Biến read\_count theo dõi bao nhiêu tiến trình hiện đang đọc đối tượng. Semaphore rw\_mutex được sử dụng như một semaphore loại trừ tương hỗ cho các writers. Nó cũng được sử dụng bởi các reader đầu tiên và cuối cùng mà ra vào vùng trọng điểm (critical section).

-Vấn đề về readers-writers và giải pháp của nó đã được khái quát hóa để cung cấp các khóa reader-writer trên vài hệ thống. Việc đạt được khóa reader-writer yêu cầu việc chỉ ra kiểu khóa: khóa đọc hoặc khóa ghi. Khi một tiến trình chỉ muốn đọc dữ liệu được chia sẻ, nó yêu cầu khóa reader-writer ở kiểu đọc. Một tiến trình muốn chỉnh sửa cái vùng nhớ được chia sẻ phải yêu cầu khóa ở kiểu ghi. Nhiều tiến trình được cho phép đạt được khóa ở kiểu đọc đồng thời, nhưng chỉ có 1 process có thể đạt được khóa ở trạng thái ghi, do đó truy cập độc chiếm được yêu cầu đối với các writer.

-Khóa reader-writer đặc biệt hữu dụng trong các trường hợp:

- + Trong các ứng dụng mà dễ dàng có thể nhận biết được những tiến trình nào chỉ đọc dữ liệu được chia sẻ và những tiến trình nào chỉ ghi các dữ liệu được chia sẻ.

- + Trong các ứng dụng mà có nhiều reader hơn writer. Bởi vì các khóa reader-writer thường yêu cầu nhiều chi phí để tạo ra hơn so với

semaphores và các khóa loại trừ tương hỗ (mutual - exclusion locks). Việc đồng thời cho phép nhiều reader tranh chấp tài nguyên được bao gồm trong việc thiết lập khóa reader-writer.

### 1.3. The Dining-Philosophers Problem (Vấn đề về bữa tối của các triết gia)

- Xét 5 nhà triết gia dành cả cuộc đời để suy tưởng và ăn. Các nhà triết gia này cùng sử dụng một chiếc bàn tròn được bao quanh bởi 5 ghế ngồi, với mỗi ghế của 1 nhà triết gia khác nhau. Giữa bàn có 1 bát cơm, và trên bàn được đặt 5 chiếc đũa (những chiếc đũa này được đặt giữa mỗi 2 người lần lượt). Khi một nhà triết gia suy nghĩ, cô ta sẽ không tương tác với những người còn lại. Sau 1 khoảng thời gian, một nhà triết gia cảm thấy đói và cố gắng nhặt 2 chiếc đũa gần nhất (những chiếc đũa giữa cô ấy và người ngồi bên trái và người ngồi bên phải). Một nhà triết gia chỉ được nhặt 1 chiếc đũa mỗi lần. Hiển nhiên, cô ấy cũng không được chọn chiếc đũa đã được cầm lên bởi người khác. Khi người triết gia bị đói đó có đủ 2 chiếc đũa vào cùng 1 thời điểm thì cô ấy ăn mà không đặt đũa xuống. Khi cô ấy ăn xong, cô ấy đặt cả 2 chiếc đũa xuống và lại bắt đầu trạng thái suy nghĩ.

- Vấn đề ăn tối giữa các triết gia được coi như là một vấn đề kinh điển về đồng bộ không phải bởi vì sự quan trọng thực tiễn của nó cũng như vì các nhà khoa học máy tính ghét các nhà triết gia mà bởi nó là một ví dụ của một lớp các vấn đề về điều khiển đồng thời. Nó là 1 đại diện cho việc cấp phát nhiều tài nguyên giữa các tiến trình trong 1 trạng thái không bị deadlock cũng như starvation.

#### 1.3.1 Giải pháp Semaphore

- Một giải pháp đơn giản là coi mỗi chiếc đũa như là 1 semaphore. 1 nhà triết gia cố gắng cầm 1 chiếc đũa bằng cách thực thi hàm wait() trên semaphore đó. Vùng dữ liệu chung lúc này là :

semaphore chopstick[5];

- Tất cả các phần tử của chopstick được khởi tạo với giá trị là 1. Mặc dù giải pháp này đảm bảo rằng không xảy ra việc 2 người cùng ăn đồng thời, tuy nhiên nó bị loại bỏ bởi việc nó có thể gây ra deadlock. Giả sử rằng cả 5 nhà triết gia đều đói và cùng cầm đũa lên. Tất cả các phần tử của chopstick lúc này sẽ được gán bằng 0. Khi mỗi triết gia cố gắng cầm chiếc đũa bên phải lên, cô ấy sẽ bị hoãn vô thời hạn.

- Một số biện pháp khắc phục vấn đề deadlock có thể kể ra như sau:

- + Cho phép nhiều nhất là 4 người ngồi cùng lúc.
- + Cho phép một nhà triết gia nhặt đũa chỉ nếu cả 2 cây đũa đều có sẵn (để làm điều này, cô ta phải nhặt chúng ở trong vùng tranh chấp).
- + Sử dụng một giải pháp không đồng bộ - một triết gia số lẻ sẽ cầm lên chiếc đũa trái trước rồi tới chiếc đũa bên phải, trong khi triết gia số chẵn sẽ cầm lên chiếc đũa bên phải trước rồi mới tới chiếc đũa bên trái.

### 1.3.2 Monitor Solution (Giải pháp giám sát)

Tiếp theo, chúng ta sẽ minh họa khái niệm giám sát bằng cách giới thiệu giải pháp không bị deadlock cho vấn đề triết gia ăn tối (dining-philosopher problem). Giải pháp này đưa ra một giới hạn rằng nhà triết gia chỉ cầm đũa lên nếu cả hai chiếc đũa có sẵn. Để lập trình giải pháp này, chúng ta cần phân biệt giữa ba trạng thái mà chúng ta có thể tìm thấy ở 1 nhà triết gia. Chúng ta sử dụng cấu trúc dữ liệu sau:

```
enum [THINKING, HUNGRY, EATING] state[5];
```

Triết gia  $i$  có thể đặt biến  $\text{state}[i] = \text{EATING}$  chỉ nếu 2 người ngồi cạnh cô ta đang không ăn : ( $\text{state}[(i+4) \% 5] \neq \text{EATING}$ ) và ( $\text{state}[(i+1) \% 5] \neq \text{EATING}$ ).

Chúng ta cũng cần khai báo

```
condition self[5];
```

Việc này sẽ cho phép triết gia  $i$  trì hoãn chính mình khi cô ấy đói nhưng không thể có được những chiếc đũa mà cô ấy muốn.

Chúng ta giờ đang ở trong tình huống mô tả giải pháp với vấn đề triết gia ăn tối. Sự phân phát những chiếc đũa được điều khiển bởi sự giám sát *DiningPhilosophers*. Mỗi triết gia, trước khi bắt đầu ăn, phải gọi hàm *pickup()*. Sau khi hoàn thành hàm này, nhà triết gia có thể ăn. Sau đó, nhà triết gia gọi hàm *putdown()*. Vì vậy, triết gia  $i$  có thể gọi hàm *pickup()* và *putdown()* theo thứ tự sau đây:

```
DiningPhilosophers.pickup(i);
```

...

eat

...

DiningPhilosophers.putdown(i)

Dễ dàng nhận thấy rằng giải pháp này đảm bảo rằng không có 2 triết gia cùng ăn đồng thời và sẽ không có deadlocks xảy ra.

## 2. Đồng bộ trong Kernel.

Tiếp theo chúng ta sẽ mô tả về cơ chế đồng bộ được cung cấp bởi hệ điều hành Windows và Linux. 2 hệ điều hành này cung cấp ví dụ tốt về cách tiếp cận khác nhau trong việc đồng bộ kernel, và như bạn sẽ thấy, cơ chế đồng bộ trong các hệ thống này khác nhau trong nhiều khía cạnh quan trọng.

### 2.1 Đồng bộ trong Windows:

Hệ điều hành Windows là một kernel đa luồng cung cấp hỗ trợ cho những ứng dụng thời gian thực và đa xử lý. Khi mà Windows kernel truy cập một biến toàn cục trên hệ thống đơn xử lý, nó tạm thời đánh dấu các gián đoạn (interrupt) cho các bộ quản lý gián đoạn mà cũng có thể truy cập các biến toàn cục. Hơn nữa, vì lý do hiệu quả, kernel đảm bảo rằng sẽ 1 luồng sẽ không bao giờ được ưu tiên khi đang giữ spinlock.

Về việc đồng bộ hóa bên ngoài kernel, Windows cung cấp các đối tượng điều phối (dispatcher objects). Sử dụng dispatcher object., các luồng đồng bộ theo nhiều cơ chế khác nhau, bao gồm mutex locks, semaphores, events và timers. Hệ thống bảo vệ dữ liệu được chia sẻ bằng cách yêu cầu 1 luồng để có được quyền sở hữu của một mutex để truy cập dữ liệu và giải phóng quyền sở hữu khi kết thúc. Cuối cùng, những timer được sử dụng để thông báo rằng 1 hoặc nhiều thread đã vượt quá lượng thời gian cho phép.

Các đối tượng điều phối (dispatcher objects) có thể ở trạng thái có tín hiệu hoặc trạng thái không có tín hiệu. Một đối tượng ở trạng thái có tín hiệu thì có sẵn, và 1 luồng sẽ không bị chặn khi đạt được đối tượng. Một đối tượng ở trạng thái không có tín hiệu thì không có sẵn, và 1 luồng sẽ chặn khi cố gắng đạt được đối tượng này.

Có một mối quan hệ tồn tại giữa các trạng thái của đối tượng điều phối (dispatcher object) và trạng thái của luồng. Khi mà 1 luồng chặn một đối tượng điều phối không có tín hiệu, trạng thái của nó thay đổi từ ready sang waiting, và luồng thì được đặt vào một hàng chờ cho đối tượng đó. Khi mà trạng thái của đối tượng điều phối chuyển sang có tín hiệu, kernel sẽ kiểm tra liệu có luồng nào đang đợi đối tượng đó không. Nếu có, kernel sẽ chuyển 1 luồng - hoặc có thể nhiều hơn - từ trạng thái

đợi sang trạng thái sẵn sàng, nơi mà chúng có thể tiếp tục được thực thi. Số luồng mà kernel chọn từ hàng đợi dựa vào loại đối tượng điều phối mà mỗi luồng đang đợi. Kernel sẽ chọn chỉ 1 luồng từ hàng đợi cho 1 mutex, vì 1 đối tượng mutex chỉ có thể được sở hữu bởi 1 luồng đơn. Với những đối tượng sự kiện(event object), kernel sẽ chọn tất cả các luồng đang đợi sự kiện đó.

Chúng ta có thể sử dụng 1 khóa mutex như là 1 môt minh họa cho đối tượng điều phối sự kiện và trạng thái luồng. Nếu một luồng cố gắng có được đối tượng điều phối mutex mà đang ở trạng thái không có tín hiệu, luồng đó sẽ bị ngừng và đặt trong hàng đợi cho đối tượng mutex. Khi mà mutex chuyển từ trạng thái có tín hiệu (bởi 1 luồng khác đã giải phóng cái khóa trên mutex), luồng đợi trước hàng đợi sẽ bị di chuyển từ trạng thái đợi sang trạng thái sẵn sàng và sẽ có được mutex lock.

Một đối tượng ở vùng tranh chấp là một mutex user-mode thường có thể đạt được và giải phóng mà không cần sự can thiệp của kernel. Trên một hệ thống đa xử lý, một đối tượng vùng tranh chấp sử dụng 1 spinlock trong khi đợi những luồng khác giải phóng đối tượng này. Nếu nó xoay quá lâu, thì các luồng đạt được sẽ cấp phát một mutex kernel và chiếm dụng CPU của nó. Những đối tượng vùng tranh chấp hiệu quả bởi vì kernel mutex được cấp phát chỉ khi có sự tranh chấp đối tượng này. Trên thực tế, rất ít xảy ra tranh chấp, nên tiết kiệm được đáng kể.

## 2.2 Đồng bộ trong Linux.

Linux cung cấp nhiều cơ chế đồng bộ khác nhau trong kernel. Như hầu hết các kiến trúc máy tính cung cấp các lệnh cho phiên bản nguyên tử của các phép tính toán, kỹ thuật đồng bộ đơn giản nhất trong Linux kernel là một số nguyên nguyên tử, được biểu diễn bằng cách sử dụng kiểu dữ liệu mờ (opaque data type) atomic\_t. Như cái tên đã chỉ ra, tất cả các phép toán sử dụng số nguyên nguyên tử được thực hiện mà không có gián đoạn. Để hình dung, xét một chương trình bao gồm một biến đếm số nguyên nguyên tử và một giá trị số nguyên.

```
atomic_t counter;
```

```
int value;
```

Những đoạn code sau đây mô tả độ hiệu quả của phép nguyên tử(atomic):

Các số nguyên nguyên tử thường hiệu quả trong nhiều trường hợp mà biến số nguyên (ví dụ như biến đếm) cần được cập nhật, vì các phép atomic không yêu cầu cơ chế khóa trước. Tuy nhiên, ứng dụng của chúng bị giới hạn tới những chiến thuật

này. Trong trường hợp có nhiều biến gây ra một race condition, phải sử dụng tới những công cụ khóa phức tạp hơn.

Mutex locks có sẵn trong Linux cho việc bảo vệ vùng tranh chấp trong kernel. Một tác vụ phải gọi hàm `mutex_lock()` trước khi vào vùng tranh chấp và hàm `mutex_unlock()` trước khi thoát khỏi vùng tranh chấp. Nếu mutex lock không có sẵn, một tác vụ gọi `mutex_lock()` sẽ được đặt vào trạng thái sleep(ngủ) và bị đánh thức (wake up) khi một tác vụ gọi hàm `mutex_unlock()`.

Linux cũng cung cấp cơ chế spinlocks và semaphore cho việc khóa trong kernel.

### **3 Đồng bộ POSIX (POSIX Synchronization):**

#### **3.1 POSIX Mutex Locks:**

- Mutex locks thể hiện cái kĩ thuật đồng bộ nền tảng được sử dụng với Pthreads. Một mutex lock được sử dụng để bảo vệ vùng tranh chấp - khi mà một luồng có được khóa trước khi vào vùng tranh chấp và giải phóng khóa khi thoát khỏi vùng tranh chấp. Pthreads sử dụng kiểu dữ liệu `pthread_mutex_t` cho mutex locks. Một mutex được tạo với hàm `pthread_mutex_init()`. Tham số đầu là một con trỏ tới mutex. Bằng cách cho NULL là tham số thứ 2, chúng ta khởi tạo một mutex với thuộc tính mặc định. Điều này được minh họa dưới đây:

-Mutex được lấy và giải phóng với hàm `pthread_mutex_lock()` và `pthread_mutex_unlock()`. Nếu mutex lock không có sẵn khi hàm `pthread_mutex_lock()` được gọi, thì hàm được gọi sẽ bị chặn cho tới khi gọi `pthread_mutex_unlock()`. Đoạn cod sau đây minh họa việc bảo vệ vùng tranh chấp sử dụng mutex locks:

Tất cả các hàm mutex sẽ trả về 0 khi hoạt động đúng. Nếu có lỗi, những hàm này trả về mã lỗi khác 0.

#### **3.2 POSIX Semaphores.**

##### **3.2.1 POSIX Named Semaphores.**

Hàm `sem_open()` được sử dụng để tạo và mở một semaphore POSIX được đặt tên:

Trong đoạn code trên, chúng ta đang đặt tên semaphore là SEM. 0\_CREATE chỉ ra rằng semaphore sẽ được tạo nếu nó không tồn tại sẵn. Thêm vào đó, semaphore có quyền đọc và ghi tới các tiến trình khác (qua tham số 0666) và được khởi tạo với giá trị 1.

Ưu điểm của semaphores được đặt tên là những tiến trình không liên quan có thể dễ dàng sử dụng một semaphore chung như là một cơ chế đồng bộ hóa bằng cách đơn giản là tham khảo tới tên của semaphore. Trong ví dụ trên, khi mà semaphore SEM được tạo, những lệnh tiếp theo gọi hàm sem\_open(với cùng tham số) bởi các tiến trình khác sẽ trả về mô tả tới semaphore đã tồn tại.

Có thể sử dụng các hoạt động wait() và signal() của semaphore trong POSIX thông qua các hàm sem\_wait() và sem\_post(). Dưới đây là 1 đoạn code minh họa:

Semaphores POSIX được đặt tên được cung cấp cả trong Linux và Mac OS.

### 3.2.2 POSIX Unnamed Semaphores(Semaphores Posix không được đặt tên):

Một semaphores không được đặt tên và khởi tạo bằng cách sử dụng hàm sem\_init(), có 3 tham số:

1. Con trỏ tới semaphore
2. Tham số chỉ ra mức độ chia sẻ.
3. Giá trị ban đầu của semaphore

được minh họa trong đoạn code dưới:

Bằng cách cho tham số 0, chúng ta chỉ ra rằng semaphore này có thể được chia sẻ bởi các luồng thuộc về tiến trình tạo ra semaphore.Thêm vào đó chúng ta khởi tạo semaphore với giá trị là 1.

Semaphore POSIX không được đặt tên cũng sử sem\_wait() và sem\_post() giống semaphore POSIX được đặt tên. Đoạn mã sau đây minh họa quá trình bảo vệ vùng tranh chấp bằng cách sử dụng semaphore không được đặt tên đã tạo bên trên:

Hàm `pthread_cond_wait()` được sử dụng trong việc đợi một biến điều kiện. Đoạn code dưới đây minh họa cách mà 1 luồng có thể đợi điều kiện `a == b` đúng sử dụng biến điều kiện Pthread:

Mutex lock được liên kết với biến điều kiện phải bị khóa trước khi hàm `pthread_cond_wai()` được gọi, vì nó được sử dụng để bảo vệ dữ liệu trong mệnh đề điều kiện khỏi race condition có thể xảy ra. Nếu khóa được lấy rồi gọi `pthread_cond_wait()`, lấy mutex lock và biến điều kiện như tham số. Gọi hàm `pthread_cond_wait()` giải phóng mutex lock, từ đó cho phép các luồng khác truy cập vùng dữ liệu được chia sẻ và có thể cập nhật giá trị của nó để mệnh đề điều kiện trở thành đúng.

Một luồng chỉnh sửa dữ liệu được chia sẻ có thể gọi hàm `pthread_cond_signal()`, từ đó tín hiệu một luồng đợi trên biến điều kiện. Điều này được mô tả bằng đoạn code sau:

Cần lưu ý rằng lời gọi hàm `pthread_cond_signal()` không giải phóng mutex lock. Nó là lời gọi tiếp theo tới `pthread_mutex_unlock()` mà giải phóng mutex. Khi mutex được giải phóng, luồng tín hiệu trở thành chủ sở hữu của mutex locks và trả về điều khiển từ lời gọi đến `pthread_cond_wait()`.

## 4 Đồng bộ trong Java:

### 4.1 Java Monitors (giám sát)

Java cung cấp cơ chế giám sát đồng thời cho đồng bộ luồng.

Mỗi đối tượng trong Java đều được liên kết với một khóa. Khi một phương thức được khai báo là `synchronize` (đồng bộ), gọi phương thức này yêu cầu phải có được khóa cho đối tượng. Chúng ta khai báo một phương thức `synchronized` bằng cách đặt từ khóa '`synchronized`' vào trong định nghĩa hàm, ví dụ như `hàn insert()` và `remove()` trong lớp `BoundedBuffer`,

Gọi một hàm đồng bộ yêu cầu phải có khóa của đối tượng thuộc lớp `BoundedBuffer`. Nếu khóa được sở hữu bởi luồng khác, luồng gọi phương thức đồng bộ sẽ bị chặn và

được đặt vào *entry set*. Entry set đại diện cho một tập các luồng chờ cho khóa có sẵn. Nếu khóa có sẵn khi một phương thức đồng bộ gọi, thì luồng gọi trở thành chủ sở hữu khóa của đối tượng và có thể truy cập vào phương thức. Khóa được giải phóng khi mà luồng thoát ra khỏi phương thức. Nếu entry set cho khóa không rỗng khi mà khóa được giải phóng, JVM sẽ tự động chọn một luồng trong tập này để trở thành chủ sở hữu của khóa.

Thêm vào việc có khóa, mỗi đối tượng cũng có liên kết với 1 tập đợi(wait set) bao gồm một tập các luồng. Cái wait set này ban đầu rỗng. Khi một luồng vào trong phương thức đồng bộ, nó sẽ sở hữu khóa cho đối tượng. Tuy nhiên, luồng này phải được xác định rằng nó không thể tiếp tục vì một điều kiện nhất định vẫn chưa được đáp ứng. Điều này sẽ xảy ra, ví dụ nếu như producers gọi hàm insert() và buffer đầy. Luồng này sau đó sẽ giải phóng khóa và đợi cho tới khi điều kiện cho phép nó tiếp tục được đáp ứng.

Khi một luồng gọi hàm wait(), những điều sau đây sẽ xảy ra:

1. Luồng giải phóng khóa cho đối tượng.
2. Trạng thái của luồng được đặt thành bị chặn (blocked).
3. Luồng được đặt trong wait set cho đối tượng.

Làm thế nào mà một luồng consumer báo hiệu là producer giờ có thể tiến hành? Thông thường, khi một luồng thoát phương thức đồng bộ, luồng đó chỉ giải phóng khóa được liên kết với đối tượng, có thể sẽ loại bỏ luồng khỏi entry set và cho đi quyền sở hữu của khóa. Tuy nhiên, cuối hàm insert() và remove(), chúng ta có lời gọi tới hàm notify(). Lời gọi tới hàm notify():

1. Tùy tiện lựa chọn một luồng T từ danh sách luồng trong wait set.
2. Di chuyển T từ wait set đến enry set.
3. Đặt trạng thái của T từ block trở thành runnable.

T giờ có đủ điều kiện để tranh khóa với các luồng khác. Một khi mà T đã lấy lại được điều khiển của khóa, nó trả về lời gọi wait(), nơi mà nó có thể kiểm tra giá trị của biến đếm (count) lần nữa.

Tiếp theo, chúng ta sẽ mô tả hàm wait() và notify() theo như được mô tả trong Figure 7.11. Chúng ta giả sử rằng buffer đang đầy và khóa cho đối tượng tiếp theo thì có sẵn.

Producer gọi hàm insert(), nếu khóa có sẵn, truy cập vào phương thức. Khi đã ở trong phương thức, producer quyết định rằng buffer đang đầy và gọi hàm wait(). Lời gọi hàm wait() giải phóng cái khóa cho đối tượng, đặt trạng thái producer về blocked, và đặt producer vào wait set cho đối tượng.

Consumer cuối cùng gọi và truy cập vào hàm remove(), khi mà cái khóa cho đối tượng giờ đã có sẵn. Consumer xóa các item từ buffer và gọi hàm notify(). Cho rằng consumer vẫn sở hữu khóa của đối tượng.

Lời gọi hàm notify() xóa producer từ wait set cho đối tượng, chuyển producer tới entry set và đặt trạng thái của producer là có thể chạy được.

Consumer thoát khỏi hàm remove(). Thoát khỏi hàm này giải phóng khóa cho đối tượng.

Producer sẽ lấy lại khóa. Nó tiếp tục việc thực thi từ lời gọi hàm wait(). Producer kiểm tra vòng lặp, quyết định nếu còn chỗ trống trong buffer, và tiến hành với phần còn lại của hàm insert(). Nếu không có lường nào ở trong wait set cho đối tượng, lời gọi hàm notify() bị bỏ qua. Khi producer thoát khỏi phương thức, nó trả lại khóa cho đối tượng.

Cơ chế đồng bộ bằng cách sử dụng hàm wait() và notify() đã tồn tại trong Java khi nó mới được tạo ra. Tuy nhiên, những lần cập nhật Java API sau này đã giới thiệu nhiều cơ chế linh hoạt hơn. Chúng ta sẽ khảo sát ở các phần tiếp theo.

## 4.2 Reentrant Locks

Reentrant locks có thể là cơ chế khóa đơn giản nhất trong Java API. Một reentrant locks được sở hữu bởi một luồng đơn và thường cung cấp truy cập loại trừ tương hỗ tới một dữ liệu được chia sẻ. Tuy nhiên, ReentrantLock cung cấp nhiều đặc điểm hơn nữa, ví dụ như cái đặt một tham số công bằng (*fairness*), thường gán lock cho luồng đợi lâu nhất.

Một luồng lấy được ReentrantLock lock bằng lời gọi lock(). Nếu lock có sẵn - hoặc nếu luồng gọi lock() đã sở hữu nó sẵn. Nếu lock không có sẵn, luồng được gọi bị chặn cho tới khi nó cuối cùng được gán cái lock khi chủ sở hữu của nó gọi unlock(). ReentrantLock hiện thực lock như dưới đây:

Thuật ngữ lập trình sử dụng try và finally cần giải thích 1 chút. Nếu cái lock được lấy qua hàm lock(), thì rất quan trọng để lock cũng được giải phóng 1 cách tương tự. Bằng cách cho unlock() vào ngoặc của finally, chúng ta đảm bảo rằng cái lock đó sẽ được giải phóng khi vùng tranh chấp hoàn thành hoặc nếu một ngoại lệ xảy ra trong khối try. Lưu ý rằng chúng ta không đặt lời gọi hàm lock() trong cặp ngoặc của try, vì lock() không trả về bất cứ ngoại lệ đã được kiểm tra nào. Cân nhắc chuyện gì sẽ xảy ra khi chúng ta đặt lock() bên trong cặp ngoặc của try và một ngoại không được kiểm tra khi lock() được gọi(ví dụ như OutOfMemoryError): finally bị kích hoạt để gọi unlock(), mà sau đó sẽ trả về *IllegalMonitorStateException* không được kiểm tra, vì lock không bao giờ được lấy. Cái *IllegalMonitorStateException* thay thế ngoại lệ không được kiểm tra mà xảy ra khi hàm lock() được gọi, từ đó ẩn đi lý do vì sao chương trình ban đầu failed.

### 4.3 Semaphore

Java API cũng cung cấp một semaphore đếm. Cách tạo semaphore được mô tả qua lệnh sau:

```
Semaphore(int value);
```

trong đó value đặc tả giá trị ban đầu của semaphore (được phép sử dụng giá trị âm). Hàm acquire() sẽ trả về *InterruptedException* nếu luồng đạt được bị gián đoạn. Ví dụ sau đây minh họa việc sử dụng semaphore cho mutex:

Lưu ý rằng chúng ta sẽ đạt lời gọi release() ở trong finally để đảm bảo rằng semaphore sẽ được giải phóng.

### 4.4 Biến điều kiện

Chúng ta tạo 1 biến điều kiện bằng việc tạo ra ReentrantLock và gọi hàm newCondition(), hàm này sẽ trả về một đối tượng Condition biểu diễn biến điều kiện cho ReentrantLock được liên kết. Điều này được minh họa trong đoạn code sau:

Một khi biến điều kiện đã đạt được, chúng ta có thể gọi hàm await() và signal() của nó, chức năng của nó giống với hàm wait() và signal() đã mô tả ở chương 6.

Chúng ta xem xét ví dụ sau đây:

Chúng ta cũng phải tạo một ReentrantLock và 5 biến điều kiện (biểu diễn điều kiện mà luồng đang đợi) để ra tín hiệu cho luồng là lượt tiếp theo là của ai. Điều này được mô tả bằng đoạn code bên dưới:

Khi một luồng vào hàm doWork(), nó gọi hàm await() trên biến điều kiện của nó nếu như threadNumber không bằng với turn, chỉ tiếp tục khi nó được ra hiệu bởi luồng khác. Sau khi một luồng đã hoàn thành công việc của nó, nó ra hiệu cho biến điều kiện liên kết với luồng mà có lượt tiếp theo.

Lưu ý rằng doWork() không cần thiết phải được khai báo đồng bộ, bởi vì ReentrantLock cung cấp mutex. Khi một luồng gọi hàm await() trên biến điều kiện nó giải phóng ReentrantLock được liên kết, cho phép luồng khác lấy cái lock mutex. Tương tự, khi mà signal() được gọi, chỉ có biến điều kiện được ra hiệu, lock được giải phóng bằng cách gọi hàm unlock().

## 5 Các cách tiếp cận thay thế:

### 5.1 Transactional memory (Bộ nhớ giao tiếp)

Xem xét ví dụ sau. Giả sử chúng ta có một hàm update() mà cập nhật vùng dữ liệu chia sẻ. Thường thì, hàm này phải được viết bằng cách sử dụng mutex locks (hoặc semaphore) như sau:

Tuy nhiên, sử dụng cơ chế đồng bộ như mutex locks và semaphores bao gồm nhiều vấn đề tiềm ẩn, trong đó có deadlock.Thêm vào đó, khi mà số lượng luồng tăng, những kiểu khóa truyền thống cũng không thích hợp, bởi mức độ tranh chấp giữa các luồng cho việc sở hữu lock trở nên rất cao.

Như một phương pháp thay thế phương pháp locking truyền thống, những đặc điểm mà cần ưu điểm của transactional memory có thể được thêm vào các ngôn ngữ lập trình. Ví dụ chúng ta thêm xây dựng atomic{S}, thứ mà đảm bảo rằng hoạt động của S thực thi như một giao dịch. Điều này cho phép chúng ta viết lại hàm update() như sau:

Lợi ích của việc sử dụng cơ chế như vậy hơn là sử dụng locks là hệ thống transactional memory - không phải người lập trình - chịu trách nhiệm cho việc đảm bảo tính nguyên tử (atomicity). Hơn nữa, một hệ thống transactional memory có thể nhận dạng được những lệnh nào ở trong khối lệnh atomic.

## 5.2 OpenMP

Chúng ta sẽ xét 1 cái nhìn tổng quan của OpenMP và hỗ trợ của nó với lập trình song song trong môi trường chia sẻ bộ nhớ. Lợi ích của OpenMP(và những công cụ tương tự) là việc tạo luồng và quản lý chúng bởi thư viện của OpenMP chứ không phải trách nhiệm của người lập trình.

Một ví dụ về việc sử dụng chỉ thị dịch vùng tranh chấp, giả sử rằng biến đếm chia sẻ có thể được chỉnh sửa bằng hàm update() như dưới đây:

```
void update(int value)  
{  
    counter+=value;  
}
```

Nếu hàm update() có thể được gọi từ một khu vực song song, race condition có thể xảy ra trên biến đếm.

Chỉ thị dịch vùng tranh chấp (critical-section compiler directive) có thể được sử dụng để giải quyết race condition như đoạn code bên dưới:

Một lợi ích của việc sử dụng chỉ thị dịch vùng tranh chấp trong OpenMP là nó dễ sử dụng hơn là mutex locks tiêu chuẩn.

## 5.3 Functional Programming Languages (Ngôn ngữ lập trình hàm)

Một vài ngôn ngữ phổ biến như C,C++,Java và C# được biết tới như là ngôn ngữ lập trình thủ tục. Ngôn ngữ lập trình thủ tục được sử dụng cho việc hiện thực hóa thuật toán được dựa trên trạng thái. Trong những ngôn ngữ này, diễn biến của thuật toán quan trọng với cách hoạt động, và trạng thái được biểu diễn bởi các biến và các cấu trúc dữ liệu. Tất nhiên, trạng thái của chương trình có thể được biến đổi, khi mà những biến có thể được gán các giá trị khác nhau theo thời gian.

Với sự phát triển của lập trình song song và đồng thời cho hệ thống đa xử lý, functional programming cho các hệ thống multicore đã được tập trung phát triển hơn, cho phép thay đổi mô hình lập trình so với ngôn ngữ lập trình thủ tục. Những sự khác nhau nền tảng giữa 2 loại ngôn ngữ lập trình này là functional programming không duy trì trạng thái. Có nghĩa là, khi một biến được khai báo và gán giá trị, giá trị của nó không thể thay đổi được nữa. Bởi vì functional languages không cho phép thay đổi trạng thái, chúng không cần quan tâm tới các vấn đề như race condition và deadlocks. Về bản chất, hầu hết các vấn đề được thể hiện trong chương này không tồn tại trong functional languages.

## 6 Summary (Tóm tắt)

- Những vấn đề cơ bản về đồng bộ như bounded-buffer, readers-writers, và dining-philosophers. Giải pháp cho các vấn đề này được phát triển sử dụng những công cụ được giới thiệu ở chương 6, bao gồm mutex locks, semaphores, monitors và các biến điều kiện.
- Windows sử dụng các đối tượng điều phối để hiện thực những công cụ đồng bộ tiến trình.
- Linux sử dụng nhiều cách tiếp cận để tránh race conditions, bao gồm biến nguyên tử, spin locks và mutex locks.
- POSIX API cung cấp mutex locks, semaphores, biến điều kiện, POSIX cung cấp 2 dạng semaphore: được đặt tên và không được đặt tên. Nhiều tiến trình không liên quan có thể dễ dàng truy cập semaphore được đặt tên đơn giản bằng cách tham khảo đến tên của nó. Semaphore không được đặt tên không thể được dễ dàng chia sẻ, và yêu cầu đặt semaphore trong vùng nhớ chia sẻ.
- Java có một thư viện đa dạng và các API cho đồng bộ. Những công cụ có sẵn bao gồm monitors(cung cấp ở mức độ ngôn ngữ) cũng như reentrant locks, semaphores, và các biến điều kiện(được hỗ trợ bởi API).
- Những cách tiếp cận thay thế để giải quyết vấn đề tranh chấp bao gồm transactional memory, openMP, và functional languages.

## 7. Câu hỏi trắc nghiệm

Câu 1: Các phương pháp Linux dùng để giải quyết race condition:

1. race condition.
2. atomic variables.
3. spin locks
4. Tất cả đáp án trên

Đáp án là D.

Câu 2: Tình huống khi có nhiều quá trình truy cập đồng thời dữ liệu chia sẻ và kết quả cuối cùng của dữ liệu phụ thuộc vào thứ tự thực thi của các quá trình gọi là :

1. race condition
2. dynamic condition
3. critical condition
4. essential condition

Đáp án là A.

Câu 3: Loại trừ tương hỗ (mutual exclusion) có thể được giải quyết bằng:

1. Không câu nào đúng
2. binary semaphore
3. mutex locks
4. cả hai mutex locks và binary locks

Đáp án là D.

Câu 4: Những công cụ nào dưới đây được dùng để đồng bộ quá trình:

1. thread
2. semaphore
3. pipe
4. socket

Đáp án là B.

Câu 5: Nếu một process đang thực thi ở vùng tranh chấp (critical section) thì không quá trình nào được phép vào thực thi trong vùng tranh chấp nữa. Điều kiện này được gọi là:

1. Loại trừ tranh chấp
2. Loại trừ tương hỗ

3. Loại trừ đồng bộ
4. Loại trừ bất đồng bộ.

Đáp án là B.

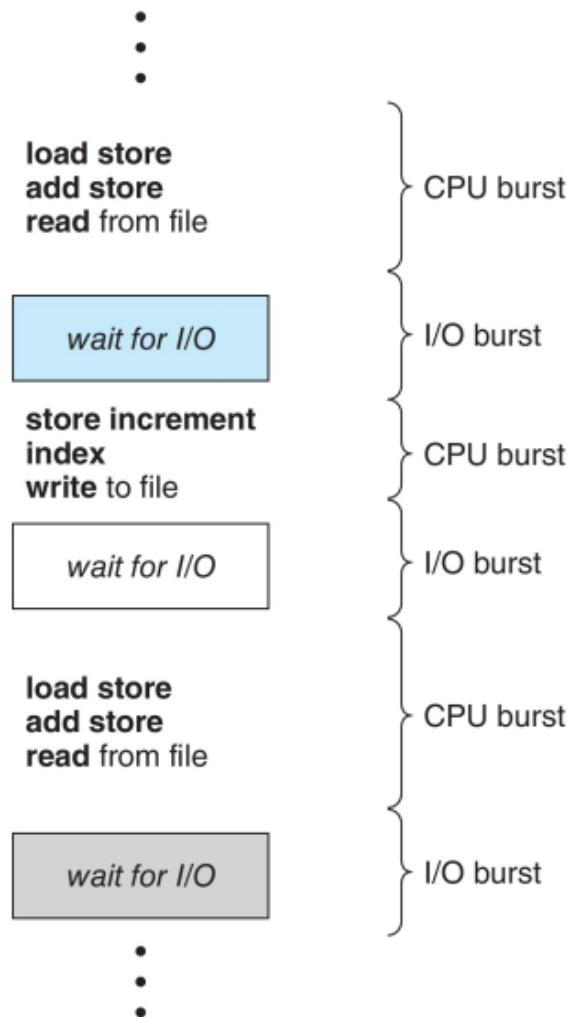
# CHƯƠNG 7: ĐỊNH THỜI CPU

## 1. CÁC KHÁI NIỆM CƠ BẢN

Định thời CPU là việc gán các job (process) lên các bộ xử lý (processor) để tối ưu hoá hiệu suất CPU.

### 1.1 CPU-I/O Burst Cycle

- Hầu hết các quá trình thực thi là sự thay thế hai trạng thái trong một chu kỳ: CPU burst và I/O burst
  - + CPU burst: thực hiện tính toán
  - + I/O burst: chờ chuyển dữ liệu vào hoặc ra hệ thống
- Một quá trình có CPU burst chiếm đa số thời gian thực thi => CPU bound. Ngược lại, đó là các quá trình I/O bound.



### 1.2 CPU Scheduler

- CPU Scheduler chọn một quá trình trong ready queue, và cấp phát CPU.

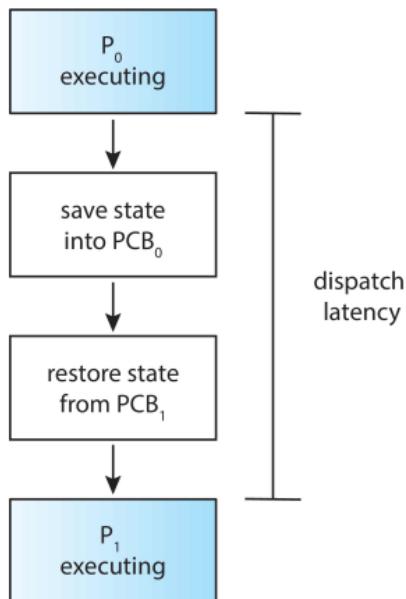
- Ready Queue có thể được lưu trữ dưới dạng FIFO queue, linked list, hàng đợi ưu tiên, ...

### 1.3 Định thời nhường

- Quyết định định thời CPU diễn ra theo một trong bốn điều kiện sau:
  - + Khi một quá trình chuyển từ trạng thái running sang waiting
  - + Khi một quá trình chuyển từ trạng thái running sang ready
  - + Khi một quá trình chuyển từ trạng thái waiting sang ready
  - + Khi một quá trình kết thúc
- **Định thời dưới điều kiện 1 và 4: không nhường (non-preemptive hoặc cooperative).** Khi ở trạng thái running, process sẽ thực thi cho đến khi kết thúc hoặc bị blocked do yêu cầu I/O.
- **Định thời dưới điều kiện 2 và 3:** process đang thực thi có thể bị ngắt quãng và chuyển về trạng thái ready, nhường CPU cho một quá trình khác (**preemptive**)
- Preemptive Scheduling có thể gây ra race conditions, cần phải xem xét các vấn đề sau:
  - + Hai process cùng truy xuất shared data
  - + Preemptive trong kernel mode
  - + Vấn đề ngắt quãng trong các hoạt động của hệ điều hành

### 1.4 Dispatcher

- Dispatcher là module đưa quyền điều khiển CPU cho process được chọn bởi short-term scheduler, bao gồm:
  - + Chuyển ngữ cảnh
  - + Chuyển sang user mode
  - + Nhảy đến địa chỉ người sử dụng tiếp tục chương trình
- Dispatch latency: thời gian dispatcher dừng một quá trình và bắt đầu chạy một quá trình khác



## 2. TIÊU CHÍ ĐỊNH THỜI

- Hiệu suất CPU (CPU utilization): giữ cho CPU luôn bận (luôn thực thi process) nếu có thể.
- **Throughput:** số lượng process hoàn thành việc thực thi trong một đơn vị thời gian.
- **Turnaround time:** Khoảng thời gian cần để việc thực thi một process hoàn tất.
- **Waiting time:** khoảng thời gian mà process phải chờ trong ready queue.
- **Response time:** khoảng thời gian process từ lúc nhận yêu cầu đến khi yêu cầu được đáp ứng lần đầu tiên
- Tối ưu hoá việc định thời là làm max CPU utilization, max Throughput, min Turnaround time, min Waiting time, min Response time
- Thông thường, cần tối ưu hoá giá trị đo trung bình của các tiêu chí này. Nhưng trong một số trường hợp, chúng ta làm max hoặc min của một giá trị đó nào đó.

## 3. GIẢI THUẬT ĐỊNH THỜI

### 3. 1 First-Come, First-Serve

- Là giải thuật **non-preemptive**
- Hàng đợi Ready là kiểu FIFO, process nào vào trước thì được ưu tiên chọn thực thi trước.
- Ví dụ:

Process    Burst Time (ms)

$P_1$                   24

$P_2$                   3

$P_3$                   3

Giả sử các quá trình đều đến tại thời điểm 0 theo thứ tự  $P_1, P_2, P_3$

Biểu đồ Gantt:



Waiting time:  $P_1 = 0, P_2 = 24, P_3 = 27$

Waiting time trung bình:  $(0 + 24 + 27)/3 = 17$

- Tác dụng phụ: Khi chúng ta có một quá trình hướng xử lý CPU (CPU-bound) và nhiều quá trình hướng nhập/ xuất (I/O bound) đến sau. Các quá trình I/O bound phải chờ và thiết bị I/O ở trạng thái rảnh. Khi quá trình CPU-bound kết thúc, các quá trình I/O bound có chu kỳ CPU rất ngắn, nhanh chóng thực thi và trợ về hàng đợi I/O, lúc này CPU rảnh → Việc sử dụng thiết bị và CPU thấp

### 3.2 Shortest-Job-First (SJF)

- Là giải thuật non-preemptive, process nào có độ dài CPU burst kế tiếp nhỏ nhất sẽ được chọn thực thi. Khi hai process có cùng CPU burst, chọn theo FCFS
- SJF được chứng minh là giải thuật tối ưu thời gian chờ, nhưng yêu cầu là phải tính được CPU-burst của các quá trình.
- Ví dụ:

Process                  Burst Time (ms)

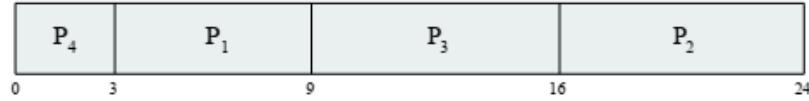
$P_1$                   6

$P_2$                   8

$P_3$                   7

$P_4$                   3

Biểu đồ Gantt:

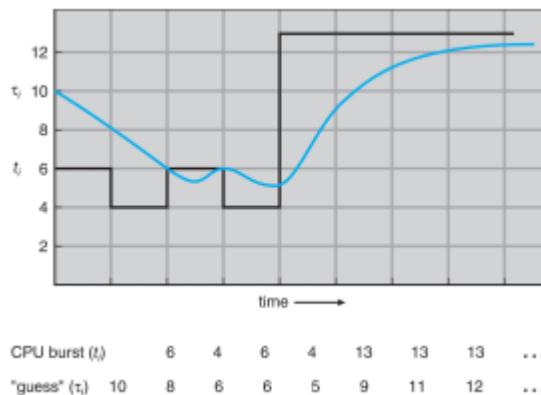


Thời gian đợi trung bình:  $(0 + 3 + 9 + 16)/4 = 7$

- Độ dài của CPU-burst chỉ có thể ước tính xấp xỉ sử dụng trung bình hàm mũ dựa vào CPU-burst của quá trình trước:  
 $t_n$ : độ dài thực của CPU-Burst thứ n  
 $\tau_{n+1}$ : giá trị dự đoán cho độ dài CPU-burst tiếp theo (thứ  $n + 1$ )  
 $\alpha, 0 \leq \alpha \leq 1$ , điều khiển sự tương quan giữa lịch sử quá khứ và lịch sử gần đây trong việc dự đoán.  
Độ dài ước tính CPU-burst thứ  $n + 1$ :

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \tau_n$$

- Ví dụ: với  $\alpha = 1/2$  và  $\tau_0 = 10$



- Giải thuật SJF có thể gây ra starvation: các quá trình có CPU-burst có thể không được thực thi khi có nhiều quá trình CPU-burst nhỏ hơn nằm trong hàng đợi.

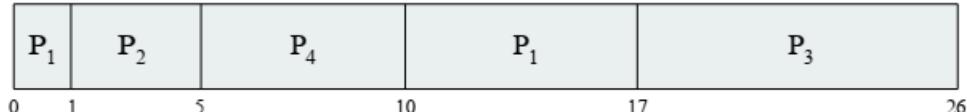
### 3.3 Shortest-Remaining-Time-First (SRTF)

- Tương tự như SJF nhưng là **preemption**

Process	Arrival Time	Burst Time (ms)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- Ví dụ: P4

Biểu đồ Gantt:



Waiting time trung bình:  $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)] / 4 = 6.5 \text{ ms}$

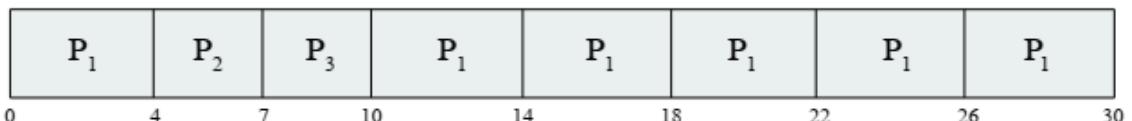
### 3.4 Round Robin (RR)

- Gần giống như FCFS, nhưng là **preemptive**

- Mỗi process có một khoảng đơn vị thời gian CPU (time quantum q) , thường là 10-100 ms để thực thi. Sau khoảng thời gian này, process bị tước quyền, nhường CPU cho process khác và được trả về cuối hàng đợi ready.
- Nếu có n quá trình trong hàng đợi ready và khoảng quantum time là q thì không có quá trình nào phải chờ đợi quá  $(n - 1) * q$  đơn vị thời gian.
- Quantum time phải lớn hơn thời gian để xử lý clock interrupt (timer) và thời gian dispatching.
- Hiệu suất:
  - + Nếu q lớn: RR trở thành FCFS
  - + Nếu q nhỏ: q phải đủ lớn, vì q nhỏ sẽ sinh ra chi phí chuyển ngữ cảnh lớn.
- Ví dụ:

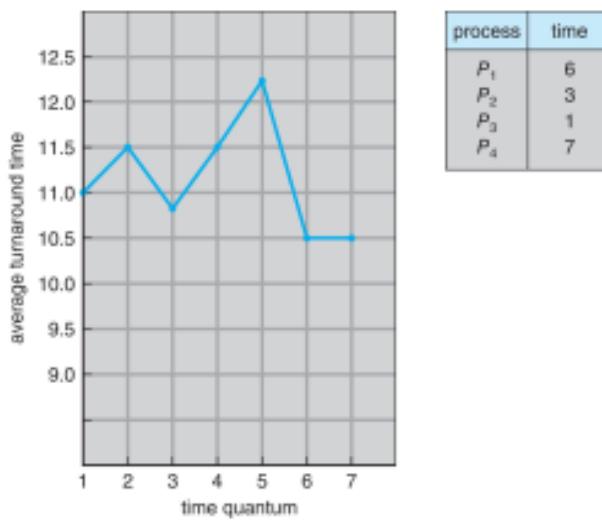
<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

Với q = 4, ta có sơ đồ Gantt:



Waiting time trung bình:  $((10 - 4) + 4 + 7) / 3 = 5,67$  ms

- Thời gian chờ đợi trung bình của RR lớn hơn SJF nhưng có thời gian đáp ứng tốt hơn
- Trong thực tế, q từ 10ms đến 100ms, thời gian chuyển ngữ cảnh thường < 10ms.
- Mối quan hệ giữa turnaround time và quantum time:



### 3.5 Priority Scheduling

- Là trường hợp tổng quát của SJF, mỗi job được gán một số ưu tiên (priority), job nào có độ ưu tiên cao hơn thì được thực thi trước.
- SJF là giải thuật được thực thi với độ ưu tiên là đảo ngược của CPU-burst time dự đoán
- Priority có thể là preemptive hoặc non-preemptive (equal-priority)
- Ví dụ:

Process	Burst Time	Priority
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

Biểu đồ Gantt:



Waiting time trung bình:  $(6 + 0 + 16 + 18 + 1) / 5 = 8.2 \text{ ms}$

- Giải thuật Priority có thể gây ra starvation, các quá trình với độ ưu tiên thấp có thể chờ vô thời hạn.
  - + Một trong những cách có thể giải quyết vấn đề này là aging, độ ưu tiên của các job sẽ tăng dần theo thời gian đợi.

### 3.6 Multilevel Queue

- Hàng đợi ready được chia thành nhiều hàng đợi với các độ ưu tiên khác nhau, mỗi hàng thực hiện các giải thuật định thời khác nhau tùy thuộc vào đặc điểm và yêu cầu định thời của process, foreground và background của process.



•  
•  
•

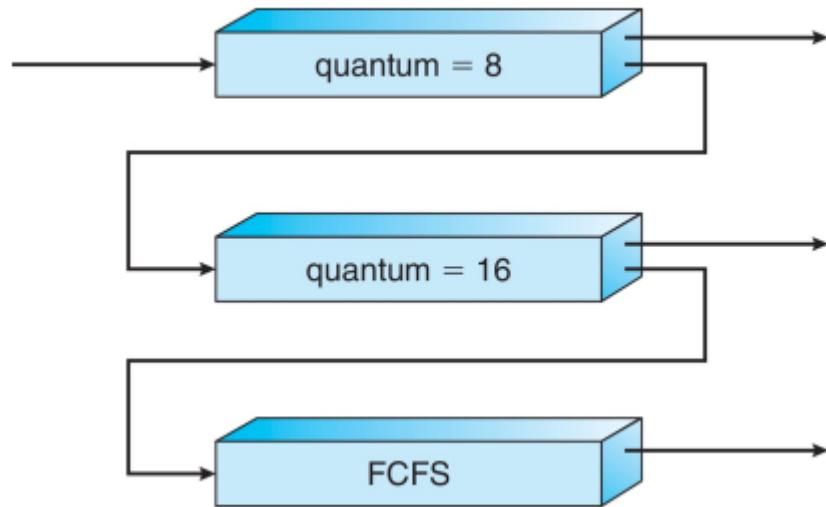


- Process không thể từ hàng đợi này sang hàng đợi khác, các quá trình này ở vĩnh viễn một hàng đợi cho đến khi thực hiện xong.
- Giải thuật này có thể gây ra starvation, các process ở độ hàng đợi có độ ưu tiên thấp có thể chờ vô thời hạn.

### 3.7 Multilevel Feedback Queue

- Gần giống như Multilevel Queue nhưng giải thuật này cho phép process di chuyển giữa các hàng đợi.
- Multilevel Feedback Queue được định nghĩa bởi các tham số sau:
  - + Số lượng hàng đợi
  - + Giải thuật định thời cho mỗi hàng đợi
  - + Phương thức upgrade hoặc demote các quá trình từ hàng đợi này sang hàng đợi khác
  - + Phương thức quyết định hàng đợi nào được thực thi.
- Cơ chế này giúp các process interactive và I/O bound ở hàng đợi có độ ưu tiên cao, một process chờ đợi lâu ở hàng đợi có độ ưu tiên thấp chuyển sang hàng đợi có độ ưu tiên cao hơn
- Ví dụ: 3 hàng đợi
  - + Q0 : thực hiện giải thuật RR với  $q = 8\text{ms}$
  - + Q1: thực hiện giải thuật RR với  $q = 16\text{ms}$

- + Q2: thực hiện giải thuật FCFS



#### 4. ĐỊNH THỜI THREAD

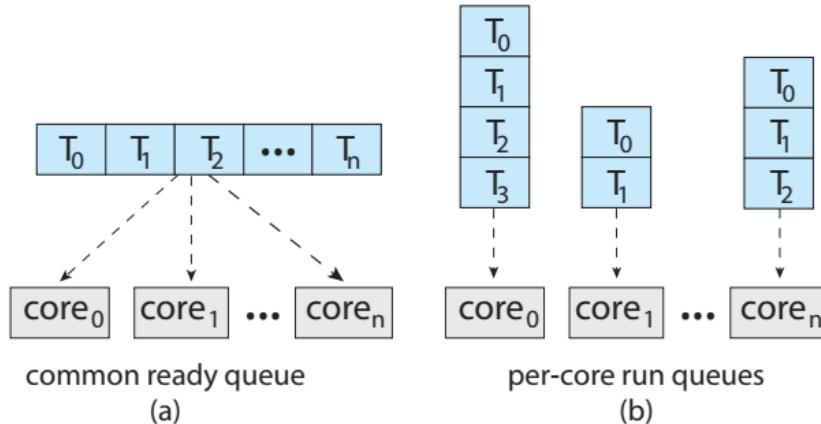
- Định thời các quá trình chỉ trong kernel thread. User thread được map với kernel thread bởi các thư viện thread. Hệ điều hành không biết gì về chúng.
- Contention Scope chỉ cho phạm vi mà các luồng cạnh tranh cho việc sử dụng CPU vật lý
- Đối với các hệ thống thực thi many-to-one và many-to-many, Process Contention Scope (PCS), được áp dụng, vì có sự cạnh tranh giữa các thread của cùng một quá trình.
- System Contention Scope (SCS), định thời thread trên kernel mode cùng chạy trên một hay nhiều CPU. Hệ thống thực thi one-to-one threads chỉ sử dụng SCS.
- Thư viện Pthread cung cấp một số contention scope: PTHREAD\_SCOPE\_PROCESS sử dụng PCS, PTHREAD\_SCOPE\_SYSTEM sử dụng SCS.

#### 5. ĐỊNH THỜI CHO HỆ THỐNG MULTI-PROCESSOR

- Với hệ thống đa xử lý, việc định thời sẽ phức tạp hơn. Lúc này, sẽ có nhiều hơn một CPU cần được giữ bận và có hiệu quả sử dụng mọi lúc.
- Load Balancing xoay quanh việc cân bằng tải giữa nhiều bộ xử lý.
- Các hệ thống đa xử lý có thể có các kiến trúc: Multicore CPUs, Multithreaded cores, NUMA systems, Heterogeneous multiprocessing (khác loại CPU)

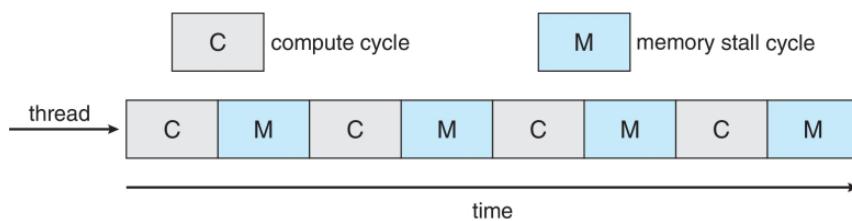
## 5.1 Tiếp cận định thời cho hệ thống multi-processor

- Đối với hệ thống : Asymmetric multiprocessor: một master processor thực hiện định thời cho các processor còn lại.
- Symmetric multiprocessor (SMP) mỗi processor có một hàng đợi riêng và bộ định thời riêng. Có 2 chiến lược để định thời các thread:
  - + Hoặc có một hàng đợi chung cho tất cả các processors
  - + Mỗi processor có hàng đợi riêng cho các thread.

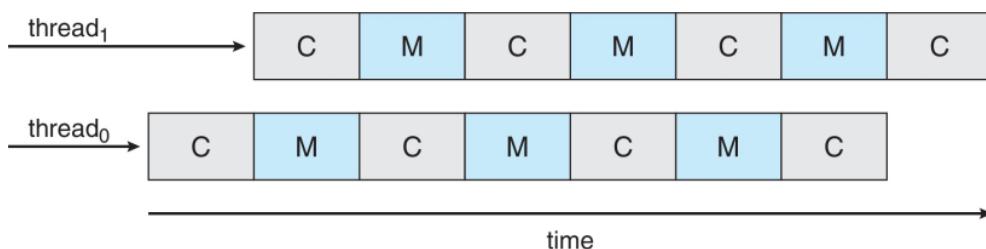


## 5.2 Multicore Processor

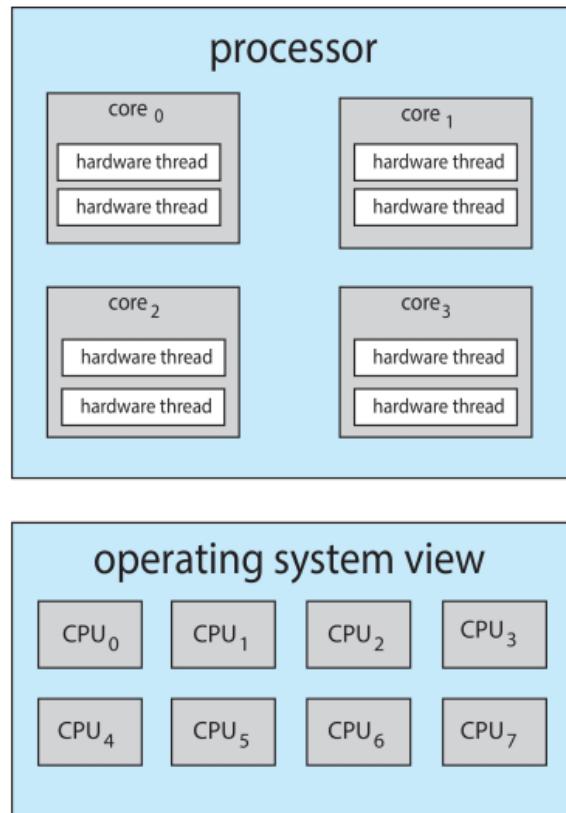
- Xu hướng hiện nay là đặt nhiều CPU trên một con chip, làm tăng tốc độ và tiêu tốn ít năng lượng hơn.
- Chu kỳ tính toán có thể bị chặn bởi thời gian cần thiết để truy cập bộ nhớ, bất cứ khi nào dữ liệu cần thiết chưa có bộ nhớ đệm



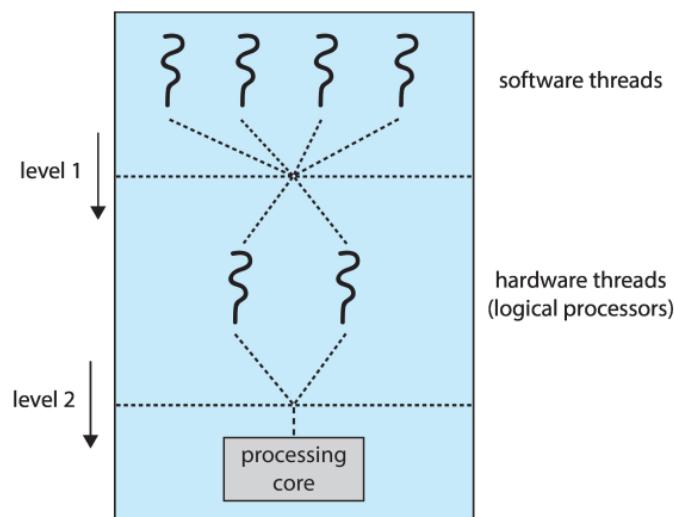
- Khi một thread xảy ra memory stall, chuyển sang thực hiện thread khác



- Chip-multithreading (CMT) gán mỗi core cho multiple hardware threads. Mỗi hardware thread duy trì kiến trúc như con trỏ lệnh, tập thanh ghi.



- Với định thời 2 cấp:
  - + Hệ điều hành quyết định software thread nào chạy trên logical CPU
  - + Core quyết định thread nào chạy trên physical core.



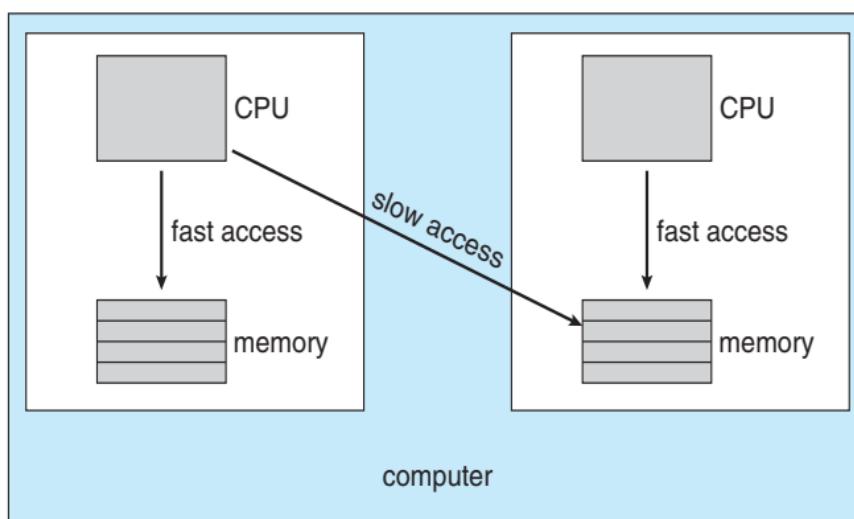
### 5.3 Cân bằng tải

- Nếu multiprocessor là SMP, tất cả các CPU đều được load để hiệu quả cao hơn
- Cân bằng có thể đạt được thông qua push migration hoặc pull migration:

- + Push migration: gồm các quá trình riêng chạy theo định kỳ và di chuyển quá trình từ bộ xử lý tải nặng hơn sang bộ xử lý ít tải hơn.
- + Pull migration: các quá trình rỗi (idle) từ hàng đợi ready sang các bộ xử lý khác
- + Push và pull không loại trừ tương hỗ.

#### 5.4 Processor Affinity

- Bộ xử lý chứa bộ nhớ đệm, giúp tăng tốc truy cập lặp lại đến cùng một vị trí bộ nhớ.
- Nếu một quá trình chuyển từ bộ xử lý này sang bộ xử lý khác → một time slice, dữ liệu trong bộ nhớ đệm sẽ bị vô hiệu hóa và load lại từ bộ bộ nhớ chính, do đó làm giảm lợi ích của bộ nhớ đệm.
- Do đó, hệ thống SMP cố gắng giữa các tiến trình trên cùng một processor thông qua processor affinity.
  - + Soft affinity: hệ điều hành giữ thread chạy trên cùng bộ xử lý nhưng không chắc chắn
  - + Hard affinity: cho phép quá trình chỉ định tập hợp bộ xử lý mà nó có thể chạy trên đó.
- Nếu hệ điều hành là NUMA-aware, nó sẽ gán bộ nhớ gần đến CPU thread đang chạy



## 6. CÂU HỎI TRẮC NGHIỆM

Câu 1: Định thời CPU được dùng để:

- A. Tăng CPU utilization
- B. Giảm throughput

- C. Tăng turnaround time
- D. Cả 3 câu trên đều đúng

Đáp án A: Định thời CPU giúp tăng CPU utilization, tăng throughput và giảm turnaround time.

**Câu 2:** Trong môi trường tương tác như hệ thống time-sharing, yêu cầu chính là cung cấp thời gian đáp ứng (response time) hợp lý, để chia sẻ tài nguyên hệ thống một cách công bằng. Giải thuật định thời được áp dụng là:

- A. Shortest-Remaining-Time-First ()
- B. Priority Scheduling
- C. Round Robin (RR)
- D. Cả 3 đáp án đều sai

Đáp án C: Giải thuật RR cho response time tốt, các quá trình chỉ thực thi trong khoảng thời gian cố định  $q \Rightarrow$  Thích hợp cho môi trường interactive.

**Câu 3:** Trường hợp nào sau đây xảy ra **định thời không nhường** (non-preemptive)?

- A. Quá trình chuyển từ running sang ready
- B. Quá trình chuyển từ running sang waiting
- C. Quá trình kết thúc
- D. Cả B và C đều đúng.

**Câu 4:** Mệnh đề nào sau đây đúng?

- I. Shortest Remaining Time First có thể gây starvation
- II. Giải thuật nhường (preemptive) có thể gây ra starvation
- III. Round Robin tốt hơn First Come First Serve về thời gian phản hồi (response time)

- A. Chỉ I
- B. I và III
- C. II và III
- D. I, II và III

Cho bảng quá trình sau, trả lời các câu hỏi từ câu 5 đến câu 7.

Process	Arrival Time	Burst Time	Priority
P1	0	10	3

P2	2	29	2
P3	3	3	1
P4	5	7	3
P5	6	12	0

Câu 5: Chọn câu trả lời đúng

- A. Waiting time trung bình theo giải thuật FCFS là 24,8 ms
- B. Waiting time của P1 theo giải thuật RR với quantum time  $q = 10$  ms là 6s
- C. Waiting time trung bình theo giải thuật SJF là 15 ms
- D. Cả 3 câu trên đều đúng

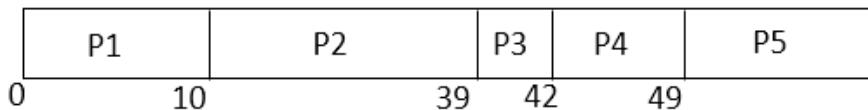
Câu 6: Waiting time trung bình với giải thuật preemptive Priority kết hợp với RR ( $q=5$ ms) là:

- A. 48,4 ms
- B. 22,8 ms
- C. 25,4 ms
- D. Đáp án khác

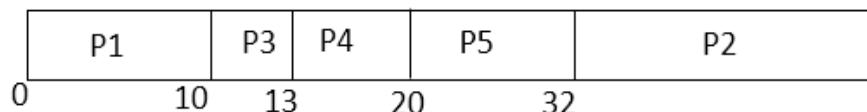
Câu 7: Turnaround time trung bình với giải thuật SRTF là:

- A. 20,6 ms
- B. 21,5 ms
- C. 23,2 ms
- D. Đáp án khác

Sơ đồ Gantt cho giải thuật FCFS



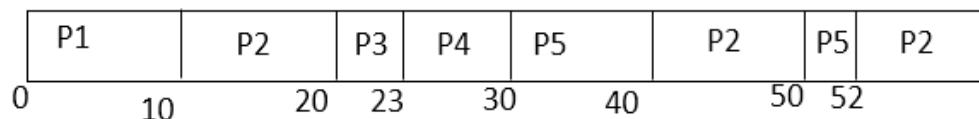
Sơ đồ Gantt cho giải thuật SJF



Sơ đồ Gantt cho giải thuật SRTF

P1	P3	P1	P4	P5	P2
0	3	6	13	20	32

Sơ đồ Gantt cho giải thuật RR ( $q = 10\text{ms}$ ):



Sơ đồ Gantt cho giải thuật preemptive Priority:

P1	P3	P5	P2	P1	P4	P1	P4
0	3	6	18	47	52	57	59

Câu 8: Những câu nào sau đây không được định thời bởi kernel?

- A. thread ở level kernel
- B. **thread ở user level**
- C. process
- D. Cả 3 câu đều sai.

Đáp án: B Thread ở user level được quản lý bởi thư viện thread và kernel không biết về các thread ở user level này.



# CHƯƠNG 8: DEADLOCKS

## I. MÔ HÌNH HỆ THỐNG (SYSTEM MODEL).

- Một hệ thống có thể được mô hình thành một tập hợp các tài nguyên hữu hạn, có thể được phân chia thành các loại khác nhau, để phân bổ cho một số tiến trình, mỗi tiến trình có nhu cầu khác nhau.
- Các loại tài nguyên có thể bao gồm: CPU cycles, file, thiết bị I/O...
- Bình thường, một tiến trình phải yêu cầu tài nguyên trước khi sử dụng và phải giải phóng nó khi hoàn thành theo trình tự sau:
  1. Yêu cầu (request). Nếu yêu cầu không thể được cấp ngay lập tức thì phải đợi cho đến khi các tài nguyên sẵn sàng.
  2. Sử dụng (use). Tiến trình sử dụng tài nguyên.
  3. Giải phóng (release). Tiến trình giải phóng tài nguyên để sử dụng cho những tiến trình khác.
- Một tập hợp các tiến trình trong trạng thái deadlock khi mọi tiến trình trong tập hợp đang chờ một tài nguyên hiện được cấp phát cho một tiến trình khác trong tập hợp.

## II. ĐẶC ĐIỂM DEADLOCK (DEADLOCK CHARACTERIZATION).

### 1. Những điều kiện cần thiết gây ra deadlock (Necessary Conditions).

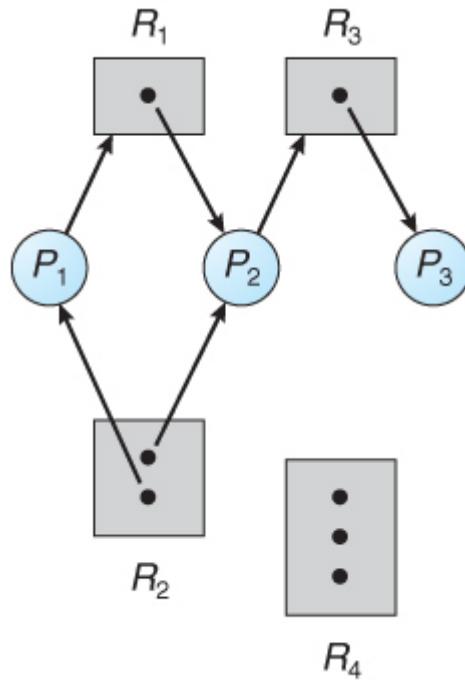
Có bốn điều kiện cần thiết để gây ra deadlock:

1. Loại trừ tương hỗ (Mutual Exclusion). Ít nhất một tài nguyên phải được giữ ở chế độ không thể chia sẻ. Nếu bất kỳ tiến trình nào khác yêu cầu tài nguyên này, thì tiến trình đó phải chờ tài nguyên được giải phóng.
2. Giữ và chờ cấp thêm tài nguyên (Hold and Wait). Tiến trình phải đang giữ ít nhất một tài nguyên và đang chờ để nhận tài nguyên thêm mà hiện đang được giữ bởi tiến trình khác.
3. Không lấy lại tài nguyên đã cấp phát (No preemption). Khi một tiến trình đang giữ một tài, thì tài nguyên đó không thể bị lấy khỏi tiến trình đó cho đến khi tiến trình tự nguyện giải phóng nó khi hoàn thành tác vụ.
4. Đợi chu trình cấp phát tài nguyên (Circular Wait). Một tập hợp các tiến trình  $\{P_0, P_1, \dots, P_n\}$  đang chờ mà trong đó  $P_0$  đang chờ một tài nguyên được giữ bởi  $P_1$ ,  $P_1$  đang chờ tài nguyên đang giữ bởi  $P_2, \dots, P_{(n-1)}$  đang chờ tài nguyên đang được giữ bởi tiến trình  $P_0$ .

## 2. Đồ thị cấp phát tài nguyên (Resource – Allocation Graph).

Trong một số trường hợp, các deadlock có thể được hiểu rõ hơn thông qua việc sử dụng đồ thị cấp phát tài nguyên, có các thuộc tính sau:

- Một tập hợp các loại tài nguyên,  $\{R_1, R_2, R_3 \dots R_N\}$ , xuất hiện dưới dạng các nút vuông trên biểu đồ. Các dấu chấm bên trong các nút tài nguyên chỉ ra các thực thể cụ thể của tài nguyên. (Ví dụ: hai dấu chấm có thể đại diện cho hai máy in laser).
- Một tập hợp các tiến trình  $\{P_1, P_2, P_3 \dots P_N\}$ .
- Các cạnh yêu cầu (Request Edges) – Một tập hợp các cạnh có hướng hướng từ  $P_i$  đến  $R_j$ , chỉ ra rằng quá trình  $P_i$  đã yêu cầu  $R_j$  và hiện đang chờ tài nguyên đó sẵn sàng.
- Cạnh gán (Assignment Edges) – Một tập hợp các cạnh có hướng từ  $R_j$  đến  $P_i$  chỉ ra rằng tài nguyên  $R_j$  đã được phân bổ để xử lý  $P_i$  và  $P_i$  hiện đang giữ tài nguyên  $R_j$ .
- Lưu ý rằng cạnh yêu cầu có thể được chuyển đổi thành cạnh gán bằng cách đảo ngược hướng của cạnh khi yêu cầu được cấp. (Tuy nhiên, cũng lưu ý rằng các cạnh yêu cầu trỏ đến hộp danh mục, trong khi các cạnh gán xuất phát từ một dấu chấm cụ thể trong hộp).



Hình 1. Đồ thị cấp phát tài nguyên

Nếu một đồ thị cấp phát tài nguyên không chứa chu trình thì không có tiến trình nào trong hệ thống bị deadlock.

Nếu một đồ thị cấp phát tài nguyên có chứa các chu trình và mỗi loại tài nguyên chỉ chứa một thực thể duy nhất, thì một chu trình ngũ ý rằng một deadlock xảy ra.

Nếu một loại tài nguyên chứa nhiều hơn một thực thể, thì sự hiện diện của một chu trình trong đồ thị cấp phát tài nguyên cho thấy khả năng của deadlock nhưng chưa chắc xảy ra.

### **III. CÁC PHƯƠNG PHÁP XỬ LÝ DEADLOCK (METHODS FOR HANDLING DEADLOCKS).**

Phần lớn, chúng ta có ba cách để xử lý deadlock:

1. Ngăn chặn hoặc tránh deadlocks. Đảm bảo rằng hệ thống sẽ không bao giờ đi vào trạng thái deadlock.
2. Phát hiện deadlock và phục hồi. Hủy bỏ một tiến trình hoặc ưu tiên một số tài nguyên khi phát hiện deadlock.
3. Bỏ qua tất cả vấn đề. Nếu deadlock chỉ xảy ra mỗi năm một lần hoặc lâu hơn, có thể tốt hơn là để chúng xảy ra rồi khởi động lại khi cần thiết hơn là chịu sự giảm hiệu suất của hệ thống cho các tác vụ phòng ngừa và phát hiện deadlock. Đây là cách tiếp cận cả Windows và UNIX đều thực hiện.

### **IV. NGĂN CHẶN DEADLOCK (DEADLOCK PREVENTION).**

Deadlock có thể được ngăn chặn bằng cách ngăn chặn ít nhất một trong bốn điều kiện bắt buộc:

#### **1. Ngăn Mutual Exclusion.**

- Các tài nguyên được chia sẻ như các tập tin chỉ đọc không dẫn đến deadlock.
- Tuy nhiên, một số tài nguyên như máy in yêu cầu truy cập độc quyền bởi một tiến trình duy nhất.

#### **2. Ngăn Hold and Wait.**

- Để đảm bảo điều kiện giữ-và-chờ cấp thêm tài nguyên không bao giờ xảy ra trong hệ thống, chúng ta phải đảm bảo rằng bất cứ khi nào một quá trình yêu cầu tài nguyên, nó không giữ bất cứ tài nguyên nào khác.
- Yêu cầu tất cả các tiến trình yêu cầu cấp phát tất cả các tài nguyên cùng một lúc. Điều này có thể gây lãng phí tài nguyên hệ thống nếu một tiến trình cần một tài nguyên sớm trong quá trình thực thi và không cần một số tài nguyên khác cho đến sau này.
- Yêu cầu các tiến trình giữ tài nguyên phải giải phóng chúng trước khi yêu cầu tài nguyên mới và sau đó lấy lại tài nguyên đã giải phóng cùng

với các tài nguyên mới trong một yêu cầu mới. Đây có thể là một vấn đề nếu một quá trình đã hoàn thành một phần hoạt động bằng cách sử dụng tài nguyên và sau đó không thể cấp phát lại sau khi giải phóng.

- Một trong hai phương pháp trên có thể dẫn đến việc tiến trình có thể chờ vô thời hạn nếu yêu cầu cấp phát tài nguyên phổ biến.

### 3. Ngăn No preemption.

- Nếu một tiến trình buộc phải chờ khi yêu cầu tài nguyên mới, thì tất cả các tài nguyên khác trước đây do quy trình này nắm giữ sẽ được giải phóng hoàn toàn, (được ưu tiên), buộc quy trình này phải lấy lại tài nguyên cũ cùng với tài nguyên mới trong một yêu cầu duy nhất.
- Khi một tài nguyên được yêu cầu và không có sẵn, thì hệ thống sẽ xem những tiến trình khác hiện đang có những tài nguyên đó và bản thân chúng đang chờ một số tài nguyên khác. Nếu một tiến trình như vậy được tìm thấy, thì một số tài nguyên có thể được ưu tiên và thêm vào danh sách các tài nguyên mà tiến trình đang chờ.
- Một trong những cách tiếp cận này có thể được áp dụng cho các tài nguyên có trạng thái được lưu và khôi phục dễ dàng, chẳng hạn như thanh ghi và bộ nhớ, nhưng thường không áp dụng được cho các thiết bị khác như máy in và ổ băng từ.

### 4. Ngăn Circular Wait.

- Một cách để tránh chờ chu trình là đánh số tất cả các tài nguyên và yêu cầu các tiến trình chỉ yêu cầu tài nguyên theo thứ tự tăng (hoặc giảm) nghiêm ngặt.
- Nói cách khác, để yêu cầu tài nguyên  $R_j$ , trước tiên, một tiến trình phải giải phóng tất cả  $R_i$  sao cho  $i \geq j$ .
- Một thách thức lớn là xác định thứ tự tương đối của các tài nguyên khác nhau.

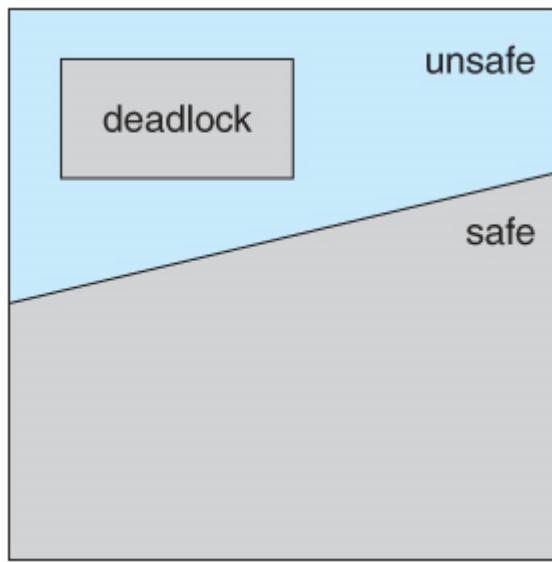
## V. TRÁNH DEADLOCK (DEADLOCK AVOIDANCE).

### 1. Trạng thái an toàn (Safe State).

- Trạng thái là an toàn nếu hệ thống có thể phân bổ tất cả các tài nguyên được yêu cầu bởi tất cả các tiến trình (tối đa đã nêu) mà không xuất hiện deadlock.
- Trạng thái sẽ an toàn nếu tồn tại một chuỗi các tiến trình an toàn  $\{P_0, P_1, P_2, \dots, P_N\}$  sao cho tất cả các yêu cầu tài nguyên cho  $P_i$  có thể được

cấp bằng cách sử dụng tài nguyên hiện được phân bổ cho  $P_i$  và tất cả các tiến trình  $P_j$  với  $j < i$ . (Tức là nếu tất cả các tiến trình trước khi  $P_i$  kết thúc và giải phóng tài nguyên, thì  $P_i$  cũng có thể hoàn thành, sử dụng các tài nguyên mà chúng đã giải phóng).

- Nếu một chuỗi an toàn không tồn tại, thì hệ thống ở trạng thái không an toàn, điều này có thể dẫn đến deadlock. (Tất cả các trạng thái an toàn đều không có deadlock, nhưng không phải tất cả các trạng thái không an toàn đều dẫn đến deadlock).



Hình 2. Không gian an toàn, không an toàn và deadlock.

## 2. Giải thuật đồ thị cấp phát tài nguyên (Resource – Allocation Graph Algorithm).

- Nếu các tài nguyên chỉ có một thực thể duy nhất thì deadlock có thể được phát hiện bởi chu trình trong đồ thị cấp phát tài nguyên.
- Trong trường hợp này, các trạng thái không an toàn có thể được nhận ra và tránh bằng cách thêm vào đồ thị cấp phát tài nguyên các cạnh thỉnh cầu (claim edges), được vẽ bằng các đường đứt nét, chỉ ra từ một tiến trình đến một tài nguyên mà nó có thể yêu cầu trong tương lai.
- Để kỹ thuật này hoạt động, tất cả các cạnh thỉnh cầu phải được thêm vào đồ thị cho bất kỳ tiến trình cụ thể nào trước khi tiến trình đó được phép yêu cầu bất kỳ tài nguyên nào. (Ngoài ra, các tiến trình chỉ có thể thực hiện các yêu cầu đối với tài nguyên mà chúng đã thiết lập các cạnh thỉnh cầu và các cạnh thỉnh cầu không thể được thêm vào bất kỳ tiến trình nào hiện đang giữ tài nguyên).

- Khi một tiến trình thực hiện một yêu cầu, cạnh thỉnh cầu  $P_i \rightarrow R_j$  được chuyển đổi thành cạnh yêu cầu. Tương tự như vậy khi một tài nguyên được giải phóng, phép gán trả lại cạnh thỉnh cầu.
- Cách tiếp cận này hoạt động bằng cách từ chối các yêu cầu sẽ tạo ra các chu trình trong đồ thị cấp phát tài nguyên.

### 3. Giải thuật của Banker (Banker's Algorithm).

- Đối với các loại tài nguyên chứa nhiều hơn một thực thể, phương pháp đồ thị cấp phát tài nguyên không hoạt động và phải chọn các phương thức phức tạp hơn (và kém hiệu quả hơn).
- Tên gọi của thuật toán này xuất phát từ các chủ ngân hàng có thể sử dụng để đảm bảo rằng khi họ cho vay tài nguyên, họ vẫn có thể đáp ứng tất cả các khách hàng của mình. (Một nhân viên ngân hàng sẽ không vay một ít tiền để bắt đầu xây nhà trừ khi họ được đảm bảo rằng sau đó họ sẽ có thể vay hết số tiền còn lại để hoàn thành ngôi nhà).
- Khi một tiến trình mới đưa vào hệ thống, nó phải khai báo số tối đa các thực thể của mỗi loại tài nguyên mà nó cần. Số này có thể không vượt quá tổng số tài nguyên trong hệ thống.
- Khi một người dùng yêu cầu tập hợp các tài nguyên, hệ thống phải xác định việc cấp phát của các tài nguyên này sẽ để lại hệ thống ở trạng thái an toàn hay không. Nếu trạng thái hệ thống sẽ là an toàn, tài nguyên sẽ được cấp; ngược lại quá trình phải chờ cho tới khi một vài quá trình giải phóng đủ tài nguyên.
- Nhiều cấu trúc dữ liệu phải được duy trì để cài đặt giải thuật Banker. Những cấu trúc dữ liệu này mã hoá trạng thái của hệ thống cấp phát tài nguyên. Gọi  $n$  là số tiến trình trong hệ thống và  $m$  là số loại tài nguyên trong hệ thống. Chúng ta cần các cấu trúc dữ liệu sau:
  - Available[m]: Một vector có  $m$  phần tử thể hiện số lượng tài nguyên của từng loại.
  - Max[n][m]: Một ma trận  $n \times m$  cho biết số lượng tối đa yêu cầu của mỗi tiến trình.
  - Allocation[n][m]: một ma trận  $n \times m$  định nghĩa số lượng tài nguyên của mỗi loại hiện được cấp tới mỗi quá trình
  - Need[n][m]: chỉ ra các loại tài nguyên cần thiết cho mỗi tiến trình. (Lưu ý rằng  $Need[i][j] = Max[i][j] - Allocation[i][j]$  với mọi  $i, j$ .)
- Để đơn giản, chúng ta quy ước kí hiệu như sau:

- Một hàng của ma trận Need,  $Need[i]$  có thể được coi là một vector tương ứng với nhu cầu của tiến trình  $i$ , tương tự cho Allocation và Max
- Ma trận  $X \leq Y$  khi  $X[i] \leq Y[i]$  với mọi  $i$ .

**a. Giải thuật phát hiện an toàn (Safety Algorithm).**

- Để áp dụng giải thuật của Banker, trước tiên chúng ta cần một thuật toán để xác định liệu một trạng thái cụ thể có an toàn không.
- Thuật toán này xác định xem trạng thái hiện tại của hệ thống có an toàn không, theo các bước sau:
  1. Đặt Work và Finish là các vector có độ dài lần lượt là  $m$  và  $n$ .
    - Work là một bản sao làm việc của các tài nguyên có sẵn, sẽ được sửa đổi trong quá trình phân tích.
    - Finish là một vector kiểu booleans, cho biết một tiến trình cụ thể có thể kết thúc.
    - Khởi tạo  $Work=Available$ ,  $Finish$  là false cho tất cả thành phần.
  2. Tìm  $i$  thỏa
    - $finish[i]==false$
    - $Need[i]<Work$
 Nếu  $i$  không tồn tại thì chuyển đến bước 4.
  3. Gán  $Work=Work+Allocation[i]$ ,  $Finish[i]=true$ . Điều này có nghĩa tiến trình  $i$  hoàn thành công việc và giải phóng tài nguyên. Sau đó lặp lại bước 2.
  4. Nếu  $Finish[i]==true$  với  $i=1, 2, \dots, n$ , thì trạng thái là an toàn.

**b. Giải thuật cấp phát tài nguyên (Resource-Request Algorithm).**

- Thuật toán này xác định xem yêu cầu mới có an toàn hay không và chỉ cấp khi nó an toàn.
- Khi một yêu cầu được đưa ra (không vượt quá các tài nguyên hiện có), giả vờ nó đã được cấp, và sau đó xem liệu trạng thái kết quả có an toàn không. Nếu an toàn, hãy cấp yêu cầu và nếu không, hãy từ chối yêu cầu, như sau:
  1. Đặt  $Request[n][m]$  cho biết số lượng tài nguyên từng loại đang yêu cầu. Nếu  $Request[i] > Need[i]$ : báo lỗi
  2. Nếu  $Request[i] > Available$  thì tiến trình đó phải chờ tài nguyên sẵn sàng. Nếu không, chuyển sang bước 3.
  3. Giả định cấp phát tài nguyên đáp ứng yêu cầu của  $P_i$  bằng cách cập nhật trạng thái hệ thống như sau:

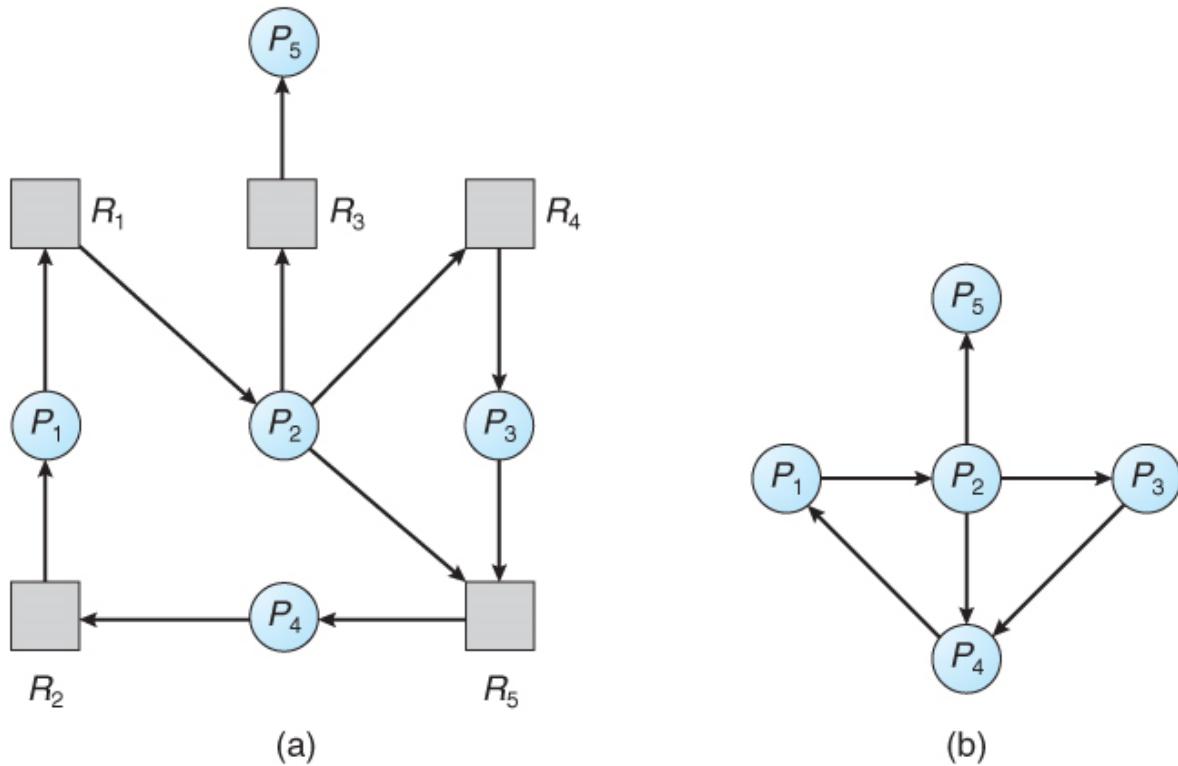
- Available=Available-Request[i]
  - Allocation[i]= Allocation[i]+Request[i]
  - Need[i]=Need[i]-Request[i]
4. Áp dụng giải thuật kiểm tra trạng thái an toàn lên trạng thái trên. Nếu trạng thái là an toàn thì tài nguyên được cấp thực sự cho Pi. Nếu trạng thái là không an toàn thì Pi phải đợi, và phục hồi trạng thái:
- Available=Available+Request[i]
  - Allocation[i]= Allocation[i]-Request[i]
  - Need[i]=Need[i]+Request[i]

## VI. PHÁT HIỆN DEADLOCK (DEADLOCK DETECTION).

- Chấp nhận xảy ra deadlock trong hệ thống, kiểm tra trạng thái hệ thống bằng giải thuật phát hiện deadlock. Nếu có deadlock thì tiến hành phục hồi hệ thống.
- Các giải thuật phát hiện deadlock thường sử dụng đồ thị cấp phát tài nguyên.
- Giải thuật phát hiện deadlock được thiết kế cho mỗi trường hợp sau:
  - Mỗi loại tài nguyên chỉ có một thực thể
  - Mỗi loại tài nguyên có thể có nhiều thực thể

### 1. Mỗi loại tài nguyên chỉ có một thực thể.

- Sử dụng wait-for graph.
- Wait-for graph được dẫn xuất từ đồ thị cấp phát tài nguyên bằng cách bỏ các node biểu diễn tài nguyên và ghép các cạnh tương ứng.
- Có cạnh từ Pi đến Pj: Pi đang chờ tài nguyên từ Pj



**Hình 3. (a) Resource allocation graph. (b) Wait-for graph tương ứng.**

## 2. Mỗi loại tài nguyên có nhiều thực thể.

- Phương pháp dùng wait-for graph không áp dụng được cho trường hợp mỗi loại tài nguyên có nhiều thực thể.
- Giả thiết: sau khi được đáp ứng yêu cầu tài nguyên, tiến trình sẽ hoàn tất và trả lại tất cả tài nguyên → giải thuật được kì vọng!
- Giải thuật phát hiện deadlock trường hợp mỗi loại tài nguyên có nhiều thực thể, các cấu trúc dữ liệu:
  - Available: vector độ dài  $m$ : số thực thể sẵn sàng của mỗi loại tài nguyên.
  - Allocation: ma trận  $n \times m$ : số thực thể của mỗi loại tài nguyên đã cấp phát cho mỗi tiến trình.
  - Request: ma trận  $n \times m$ : yêu cầu hiện tại của mỗi tiến trình.

## 3. Giải thuật phát hiện deadlock (Detection – Algorithm Usage).

1. Các biến Work và Finish là vector kích thước  $m$  và  $n$ . Khởi tạo:

Work=Available. Nếu  $\text{Allocation}[i] \neq 0$  thì  $\text{Finish}[i]=\text{false}$ , ngược lại thì  $\text{Finish}[i]=\text{true}$ .

2. Tìm  $i$  thỏa mãn:  $\text{Finish}[i]=\text{false}$  và  $\text{Request}[i] \leq \text{Work}$ . Nếu không tồn tại  $i$  như thế, đến bước 4

3. Work=Work+Allocation[i], Finish[i]=true, quay về bước 2.
4. Nếu tồn tại i với Finish[i]=false, thì hệ thống đang ở trạng thái deadlock. Hơn thế nữa, nếu Finish[i]=false thì Pi bị deadlocked.

## VII. PHỤC HỒI DEADLOCK (RECOVERY FROM DEADLOCK).

Các giải pháp khi phát hiện deadlock:

- Thông báo trực tiếp cho người vận hành hệ thống để họ can thiệp thủ công.
- Chấm dứt một hoặc nhiều tiến trình liên quan đến deadlock.
- Lấy lại tài nguyên.
  1. Phục hồi khỏi deadlock bằng cách chấm dứt tiến trình (Process Termination).
- Phục hồi hệ thống khỏi deadlock bằng cách.
  - Chấm dứt tất cả tiến trình bị deadlocked.
  - Chấm dứt lần lượt từng tiến trình cho đến khi không còn deadlock (Sử dụng giải thuật phát hiện deadlock để xác định còn deadlock hay không).
- Có nhiều yếu tố có thể đi đến quyết định tiến trình nào sẽ chấm dứt tiếp theo:
  - Độ ưu tiên của tiến trình.
  - Thời gian đã thực thi của tiến trình và thời gian còn lại.
  - Loại tài nguyên mà tiến trình đã sử dụng.
  - Tài nguyên mà tiến trình cần thêm để hoàn tất công việc.
  - Số lượng tiến trình cần được chấm dứt.
  - Process là interactive process hay batch process.

### 2. Phục hồi khỏi deadlock bằng cách lấy lại tài nguyên (Resource Preemption).

- Lần lượt lấy lại tài nguyên từ các tiến trình, cấp phát chúng cho tiến trình khác cho đến khi không còn deadlock nữa.
- Các vấn đề khi thu hồi tài nguyên:
  - Chọn “nạn nhân”: chọn tài nguyên và tiến trình nào (có thể dựa trên số tài nguyên sở hữu, thời gian CPU đã tiêu tốn...)
  - Rollback: rollback process bị lấy lại tài nguyên trở về trạng thái an toàn, rồi tiếp tục process từ trạng thái đó. Do đó hệ thống cần lưu giữ một số thông tin về trạng thái các process đang thực thi.
  - Starvation: để tránh starvation, phải bảo đảm không có tiến trình nào mà luôn bị lấy lại tài nguyên mỗi khi phục hồi khỏi deadlock.

## VIII. CÂU HỎI TRẮC NGHIỆM.

1. Điều kiện nào sau đây là cần thiết để xảy ra deadlock?

- A. Loại trừ tương hỗ (Mutual exclusion).
- B. Một tiến trình có thể giữ các tài nguyên được phân bổ trong khi chờ phân công các tài nguyên khác.
- C. Không tài nguyên nào có thể bị lấy lại khỏi tiến trình đang giữ nó.
- D. Tất cả đều đúng.

2. Giải thuật của Banker thì \_\_\_\_\_ so với giải thuật đồ thị cấp phát tài nguyên.

- A. Kém hiệu quả hơn.
- B. Hiệu quả hơn.
- C. Hiệu quả như nhau.
- D. Tất cả đều sai.

3. Một cạnh từ tiến trình Pi đến tiến trình Pj trong wait-for graph cho biết rằng:

- A. Pi đang chờ Pj giải phóng tài nguyên mà Pi cần.
- B. Pj đang chờ Pi giải phóng tài nguyên mà Pj cần.
- C. Pi đang đợi Pj kết thúc.
- D. Pj đang đợi Pi kết thúc.

4. Nếu wait-for graph chứa một chu trình thì:

- A. Deadlock không tồn tại.
- B. Deadlock tồn tại.
- C. Hệ thống trở về trạng thái an toàn.
- D. Hoặc tồn tại deadlock, hoặc hệ thống ở trạng thái an toàn.

5. Trong giải thuật của Banker, ma trận Need được tính:

- A. Allocation – Available.
- B. Max – Available.
- C. Max – Allocation.
- D. Allocation – Max.

6. Biểu đồ cấp phát tài nguyên không áp dụng cho hệ thống cấp phát tài nguyên

- A. Với mỗi loại tài nguyên có nhiều thực thể.
- B. Với mỗi loại tài nguyên có một thực thể.
- C. Với mỗi loại tài nguyên có một hay nhiều thực thể.
- D. Tất cả đều sai.

# CHƯƠNG 9: BỘ NHỚ CHÍNH

## I. Phần cứng cơ bản:

Hệ điều hành luôn đảm bảo quá trình này không thể truy xuất không gian của quá trình khác.

**Thanh ghi địa chỉ nền (Base):** thanh ghi chứa địa chỉ không gian bắt đầu của một quá trình

**Limit:** chứa độ lớn không gian của quá trình

Vậy địa chỉ truy xuất hợp lệ của quá trình từ base đến base+limit-1

### 1. Liên kết địa chỉ (Address Binding):

Địa chỉ lệnh (instruction) và dữ liệu (data): được chuyển đổi thành địa chỉ vật lý của bộ nhớ thực có thể xảy ra tại ba thời điểm khác nhau:

+ **Compile time** : Trong absolute code phải chỉ rõ biến cần lưu nằm ở vị trí nào trong bộ nhớ. Điều này dẫn đến một số bất lợi như lúc nạp địa chỉ có một quá trình khác đang chiếm vị trí vùng nhớ đó sẽ phát sinh đụng độ. Và phải biên dịch lại mỗi lần thay đổi địa chỉ.

+ **Load time** : Tại thời điểm biên dịch nếu không biết địa chỉ thực thi thời điểm loading, phải chuyển đổi địa chỉ khả tái định theo một mốc chuẩn (base address). Khuyết điểm là địa chỉ thực được tính toán lúc chương trình thực thi và tiến hành load lại nếu địa chỉ base thay đổi

+ **Execution time** : Tại thời điểm thực thi sẽ kiểm tra còn không gian nào đủ trống cho quá trình . Một quá trình đang thực thi có thể bị hoãn lại nhường cho quá trình khác và dữ liệu của nó sẽ được chuyển xuống bộ nhớ thứ cấp do không đủ bộ nhớ. Khi quay lại tiếp tục thực thi dữ liệu sẽ được chuyển lên vùng không gian trống mới khác với ban đầu. Điều này giúp không gian bộ nhớ linh hoạt hơn. Đây là lựa chọn của hầu hết các hệ điều hành phổ biến hiện nay

### 2. Không gian địa chỉ vật lý logic (Logical Versus Physical Address Space)

- **Địa chỉ luận lý (Logical Address)** : được tạo bởi CPU. Còn được gọi là địa chỉ ảo (virtual address)
- **Địa chỉ vật lý (Physical Address)** : được quản lý bởi MMU
- **Memory-management unit (MMU):** là phần cứng có nhiệm vụ ánh xạ địa chỉ luận lý sang địa chỉ vật lý

- **Không gian địa chỉ luận lí (Logical address space):** Cho tất cả các quá trình đều bắt đầu bằng 0. Nếu còn đủ không gian, hệ thống sẽ luôn tìm thấy vùng không gian vật lí để cấp phát

### 3. Nạp động (Dynamic loading):

Những thủ tục ít dùng chỉ được nạp vào bộ nhớ chính khi được gọi đến. Điều này làm tăng tính hiệu quả cho bộ nhớ vì các thủ tục ít dùng sẽ không chiếm chỗ trong bộ nhớ. Rất hiệu quả khi lưu trữ một mã chương trình lớn nhưng tần số sử dụng thấp (ví dụ các mã chương trình sửa lỗi). Thông thường người dùng sẽ chịu trách nhiệm thiết kế hiện thực các chương trình dynamic loading mà không cần sự can thiệp hệ điều hành.

## II.Phân bổ bộ nhớ liên tục (Contiguous Memory Allocation)

Bộ nhớ chính phải hỗ trợ cho cả hệ điều hành và cho các chương trình người dùng. Nếu không tìm thấy vùng không gian liên tục quá trình cần thì quá trình đó sẽ bị trì hoãn. Dẫn đến sử dụng bộ nhớ không hợp lý.

### 1. Bảo vệ bộ nhớ (Memory Protection):

Đảm bảo quá trình có thể truy cập vùng không gian nó đã được cấp phát không được truy cập vào không gian nhau của quá trình khác. Quá trình kiểm tra bằng cách so sánh địa chỉ do CPU gửi tới có thuộc [base, base limit]. The MMU sẽ gán địa chỉ đã duyệt thành công vào thanh ghi relocation, sau đó ánh xạ thành địa chỉ vật lí và chuyển cho bộ nhớ thực

### 2. Cấp phát vùng nhớ (Memory Allocation):

- **Variable-partition:** sơ đồ phân vùng chỉ ra vùng nào đã được cấp phát và vùng nào trống có sẵn
- **Hole:** một vùng lưu trữ có sẵn chưa được sử dụng. Khi một quá trình đến nó sẽ được cấp phát 1 hole thỏa mãn yêu cầu của nó
- **First fit:** chọn vùng không gian trống đầu tiên đủ với yêu cầu để cấp phát
- **Best fit :** chọn vùng không gian có độ dư thừa nhỏ nhất so với nhu cầu
- **Worst fit:** chọn vùng không gian trống lớn nhất

### 3. Phân mảnh

- **Phân mảnh ngoại (External fragmentation):** Tổng vùng không gian có thể cấp phát lớn hơn vùng không gian quá trình yêu cầu nhưng vẫn không thể cấp phát do không liên tục.
- **Phân mảnh nội (Internal fragmentation)** : Cung cấp dư thừa vùng không gian cho quá trình, vùng dư thừa không dùng để làm gì cả dẫn đến phân mảnh nội.

## III. Phân trang (Paging)

### 1. Phương pháp cơ bản:

- **Frames:** không gian vật lý của quá trình thành những frame có kích thước như nhau.
- **Pages :** không gian luân lí của quá trình được chia thành những trang nhớ có kích thước như nhau.
- **Bảng phân trang (Page table):** gồm 2 thông tin cơ bản là chỉ mục trang và chỉ mục.
- **Page number:** là chỉ số chứa địa chỉ nền của page trong bộ nhớ vật lý.
- **Page offset:** cùng với địa chỉ nền xác định địa chỉ vật lý
- **Các bước MMU ánh xạ từ địa chỉ luân lí sang địa chỉ vật lý:**
  - + Trích xuất page number và dùng nó làm chỉ mục (index) cho bảng phân trang.
  - + Trích xuất frame number từ bản phân trang.
  - + Thay page number bằng frame number

Sử dụng kỹ thuật phân trang giúp chúng ta tránh phân mảnh ngoại. Kích thước một frame bằng một page. Một page có thể được ánh xạ vào bất kì frame nào nếu frame còn trống do đó các page không cần liên tục nhau tránh xảy ra phân mảnh ngoại.

Tuy nhiên, kỹ thuật phân trang vẫn tồn tại phân mảnh nội do kích thước của một frame khá lớn và không phải lúc nào chương trình cũng cần có nhu cầu số frame chính xác.

### 2. Hỗ trợ phần cứng

Việc sử dụng con trỏ để lưu trữ bảng phân trang chỉ đạt yêu cầu khi kích thước của bảng phân trang nhỏ (khoảng 256 entries). Hầu hết các CPU hiện tại cho phép đến 20 entries. Đối với các máy như vậy thì sử dụng thanh ghi để lưu trữ nhanh hơn sẽ không hiệu quả. .

- **Thanh ghi page-table base (PTBR)** trỏ đến bảng phân trang
- **Thanh ghi page-table length (PTBL)** biểu thị kích thước của bảng phân trang
- **Mỗi lần truy cập dữ liệu cần 2 thao tác:**

- + Dùng page number làm chỉ mục để truy cập vào bảng phân trang lấy number frame
- + Dùng number frame vừa lấy được và page offset để truy cập lấy dữ liệu từ frame

**Hệ thống thường sử dụng cache có tốc độ truy xuất cao được gọi là Translation Look-aside Buffers (TLBs) (also called associative memory).** Mỗi mục (entry) trong TBL gồm 1 khóa và một giá trị( key & value). Nếu page nằm trong TBL lấy ngay được frame mà không cần truy cập bộ nhớ chính để vào bảng phân trang lấy frame. Ngược lại, vào vào bộ nhớ chính để truy cập vào bảng phân trang, sau đó page number và frame number được cập nhật vào TBL cho lần truy xuất kế tiếp.

- **wire-down** :Khi TBL đầy sẽ thay thế entry mới với entry trong danh sách “least recent use”(LRU). Một vài entry không cho phép bị thay thế trong TBL được gọi là wire-down

### 3. Thời gian truy xuất hiệu dụng (Effective memory-access time):

- **Hit ratio** : tỷ lệ tìm thấy page number trong TBL so với tổng số lần truy cập.
- **EAT=** (2-hit ratio)\*thời gian truy xuất bộ nhớ chính+ thời gian tìm kiếm trong TBL

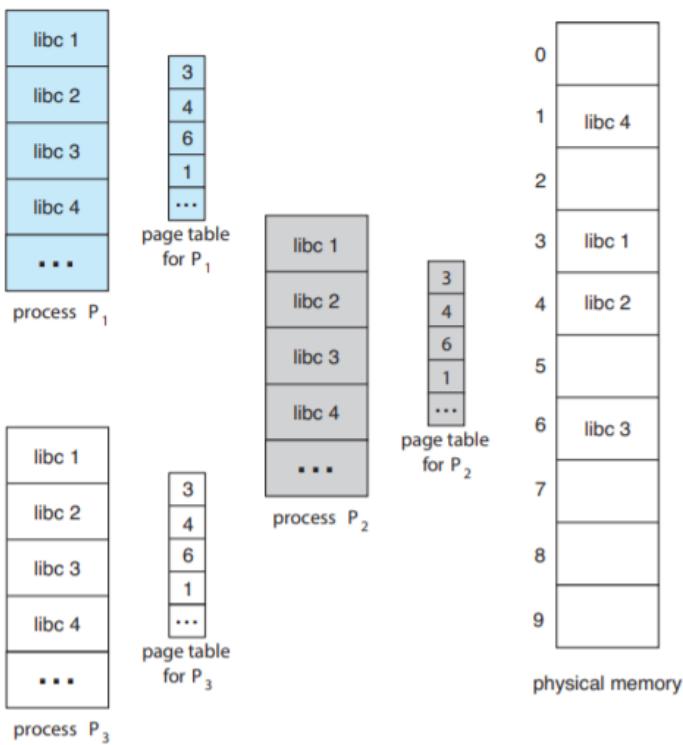
### 4. Bảo vệ

Bảo vệ bộ nhớ được thực hiện bằng cách gắn bộ nhớ với các bit bảo vệ (protection bit) được giữ trong bảng phân trang. Các bit biểu thị các thuộc tính sau: read-only, read-write, execute-only

Ngoài ra còn có một valid/invalid bit được gắn trong mỗi mục của bảng phân trang

### 5. Chia sẻ trang (Shared Page):

Bản mã chỉ có thể đọc (reentrant code) có thể được chia sẻ giữa các quá trình tương tự như các luồng chia sẻ vùng không gian với nhau



#### IV. Tổ chức bảng trang

Bảng trang thường được tổ chức theo 3 kiểu:

##### 1. Cấu trúc bảng phân trang theo kiểu kế thừa (Phân trang 2 cấp) - Hierarchical paging

Chia nhỏ không gian địa chỉ luận lý thành nhiều bảng trang hơn nữa.

###### 1.1 Bảng phân trang 2 mức:

Các hệ thống hiện đại đều hỗ trợ không gian địa chỉ ảo rất lớn (232 đến 264). Để tiện việc tìm kiếm trong bảng phân trang, lúc này bản phân trang cũng được phân trang

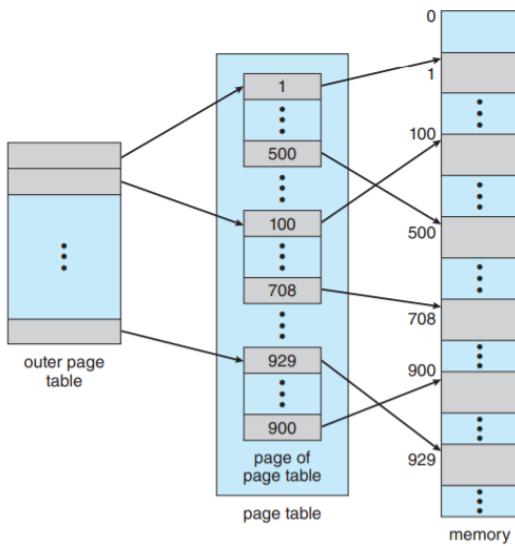


Figure 9.15 A two-level page-table scheme.

outer page	inner page	offset
$p_1$ 42	$p_2$ 10	$d$ 12

## 1.2 Bảng phân trang 3 mức:

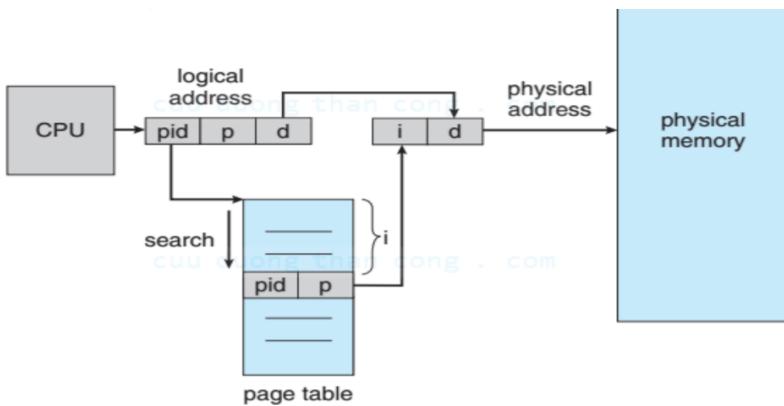
outer page	inner page	offset
$p_1$	$p_2$	$d$

2nd outer page	outer page	inner page	offset
$p_1$ 32	$p_2$ 10	$p_3$ 10	$d$ 12

## 2. Cấu trúc bảng phân trang theo kiểu nghịch đảo – Inverted page tables:

Bảng trang nghịch đảo (IBM system/38, IBM RISC, IBM RT): sử dụng cho tất cả các process

Địa chỉ luận lý  $\langle$ process-id, page-number, offset $\rangle$ .



### 3. Cấu trúc bảng phân trang theo kiểu dùng hash function- Hashed page tables

Thường dành cho hệ thống lớn hơn 32 bit. Mỗi mục của bản băm gắn với một danh sách liên kết, mỗi phần tử của danh sách bao gồm (chỉ số trang ảo và chỉ số frame). Chỉ số trang ảo được biến đổi qua hàm băm tạo thành hashed value. Hai chỉ số này được lưu vào danh sách liên kết có chỉ số hashed value

### 4. Oracle SPARC Solaris

- Có hai hàm băm một cho kernel và một cho chương trình của người dùng. Mỗi vùng là một không gian bộ nhớ ảo liên tục được ánh xạ. Mỗi vùng có địa chỉ nền và số lượng trang mà vùng đại diện (span)
- TLB giữ các mã hóa từ TTEs sang cho việc tra cứu phần cứng nhanh chóng.
- TLB tìm kiếm: nếu lỡ, vào TSB tìm TTE tiếp tục tìm. Nếu không thấy thì kernel thoát khỏi tìm kiếm trong bảng hash table. Nếu tìm thấy copy TSB vào TLB.

## V. Cơ chế Swapping:

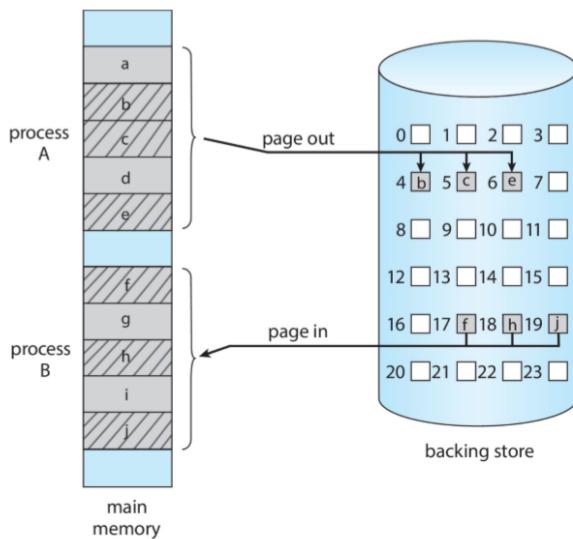
### 1. Tiêu chuẩn swapping

Một hệ thống có thể tạm thời swap ra khỏi bộ nhớ chính và lưu trên bộ nhớ phụ. Chính sách của Swapping

- **Round-robin:** swap out P1(vừa tiêu thụ hết quantum của nó), swap in P2, thực thi P3,... -
- **Roll out, roll in:** dùng trong cơ chế định thời theo độ ưu tiên (priority-based scheduling). Process có độ ưu tiên thấp sẽ bị swap out nhường chỗ cho process có độ ưu tiên cao hơn vừa đến.

## 2. Swapping với cơ chế phân trang:

Thường được sử dụng trên các hệ thống UNIX. Không được sử dụng đồng thời trong hệ điều hành.



## 3. Swapping trên hệ thống di động:

Thường không được hỗ trợ nhiều. Địa chỉ nền của flash thông lượng kém, số lượng chu kỳ hạn chế,..

### VI. Question:

**Câu hỏi 1:** Địa chỉ lệnh (instruction address) và dữ liệu nên được chuyển đổi thành địa chỉ vật lý nên ở thời điểm nào sao đây? Vì sao?

- A. Execution time vì trong thời gian thực thi hệ thống sẽ giúp chúng ta tìm vùng không gian trống đáp ứng đủ nhu cầu của quá trình để cấp phát
- B. Chuyển đổi vào load time sẽ giảm thời gian tính toán địa chỉ.
- C. Compile time là tốt nhất do chúng ta được tự ý lựa chọn địa chỉ lưu các biến trong chương trình, dễ dàng quản lý
- D. Tùy vào nhu cầu sử dụng mà chúng ta sẽ lựa chọn thời điểm.

Đáp án: A

**Câu hỏi 2:** Tại sao cơ chế phân trang paging có tồn tại trạng thái phân mảnh nội (external fragment) và phân mảnh ngoại không ? Vì sao?

- A. Kích thước một frame bằng một page. Một page có thể được ánh xạ vào bất kì frame nào nếu frame còn trống do đó các page không cần liên tục nhau tránh xảy ra phân mảnh nội.
- B. Kích thước của một frame khá lớn và không phải lúc nào chương trình cũng cần có nhu cầu số frame chính xác do đó vẫn còn không gian dư thừa do đó vẫn tồn tại phân mảnh nội.
- C. Tránh phân mảnh nội do không có phân mảnh ngoại
- D. Vẫn tồn tại phân mảnh nội

Đáp án: B

**Câu hỏi 3:** Giả sử mỗi quá trình được cấp phát không gian nhớ có kích thước 32768 bytes. Một quá trình A yêu cầu được cấp phát vùng nhớ như sau: 10284 bytes, data : 9786 bytes, stack: 11770 bytes. Hệ thống sẽ phải đáp ứng toàn bộ yêu cầu của quá trình A tại một thời điểm bao nhiêu trang nhớ để A được thực thi

- A. 7 trang
- B. 9 trang
- C. 8 trang
- D. Không đáp ứng được do số trang A yêu cầu lớn hơn số trang hệ thống có thể đáp ứng

Đáp án: D

Số trang nhớ có thể cấp phát =  $32768/4096=8$

Số trang nhớ A cần=  $(10284+9786+11770)/4096= 9\text{trang}$

**Câu hỏi 4:** Chọn nhận định đúng về các giải thuật

- A. Giải thuật worst fit tốt nhất vì cấp phát vùng không gian dư giúp chương trình chạy nhanh hơn
- B. Giải thuật best fit tiết kiệm thời gian tìm kiếm vùng không gian hơn first fit và worst fit.
- C. Best fit là một trường hợp đặc biệt của first fit
- D. Cả A,B,C đều đúng

Đáp án: C

**Câu hỏi 5:** Nhiệm vụ ánh xạ địa chỉ luận lí sang vật lí được thực hiện bởi:

- A. Các hàm xin cấp phát malloc(), alloc()
- B. C, D đúng
- C. Hệ điều hành
- D. Đơn vị quản lý bộ nhớ (Memory Management Unit)

Đáp án: B

**Câu hỏi 6:** Phát biểu nào sau đây về Translation Look-aside Buffers - TBL là đúng:

- A. TBL chỉ chứa lượng entry có number page cố định không đổi theo thời gian
- B. TBL giúp giảm thời gian truy xuất vào bộ nhớ chính để truy cập vào bảng phân trang nếu number page đang tìm có trong TBL
- C. Nếu page number không được tìm thấy trong TBL quá trình sẽ bị buộc dừng lại
- D. TBL là một phần mềm hỗ trợ tốc độ truy xuất page number

Đáp án B



# CHƯƠNG 10: BỘ NHỚ ẢO

## 1. Dẫn nhập

Yêu cầu cơ bản trong việc thực thi chương trình đó là lệnh đang được thực thi phải nằm trong vùng nhớ vật lý. Chúng ta có hướng giải quyết đầu tiên đó là đặt toàn bộ vùng nhớ luận lý sang vùng nhớ vật lý. Nhưng điều này sẽ giới hạn kích cỡ của chương trình bằng với kích cỡ của bộ nhớ chính và trên thực tế nhiều thực nghiệm đã chỉ ra rằng một chương trình không cần hoàn toàn nằm trong bộ nhớ vì vài lý do sau đây:

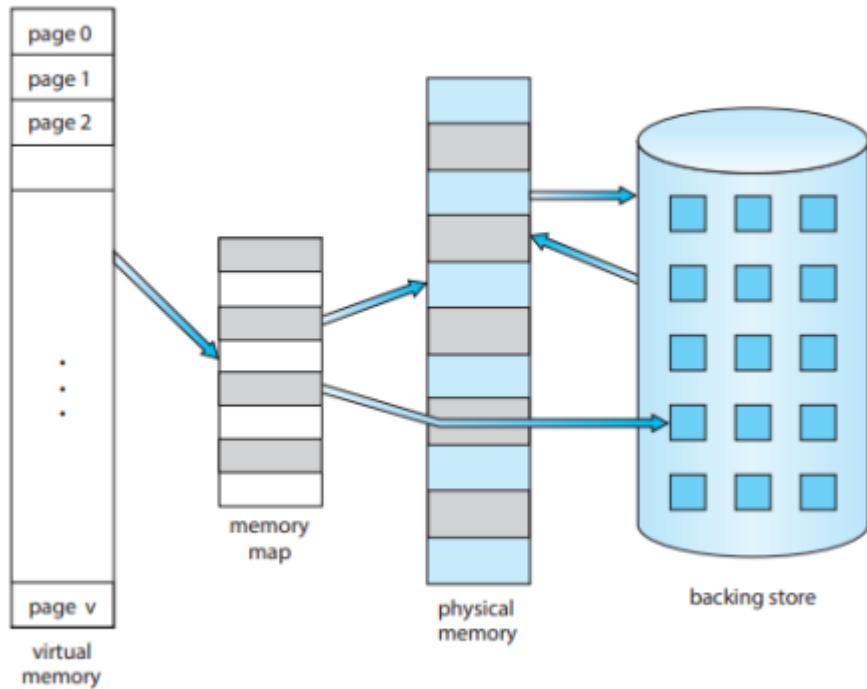
- \* Chương trình thường sẽ có những dòng lệnh xử lí ngoại lệ và đường như rất ít trường hợp xảy ra ngoài thực tế.
- \* Mảng và danh sách thường được khai báo nhiều hơn so với số lượng thực sự cần thiết.

Việc xử lí một chương trình từng phần mang lại cho chúng ta một số lợi ích:

- \* Một chương trình sẽ không còn bị hạn chế bởi số lượng vật lý bộ nhớ có sẵn. Người dùng sẽ có thể viết chương trình cho một không gian địa chỉ ảo cực lớn, đơn giản hóa công việc lập trình.
- \* Có thể chạy nhiều chương trình cùng một lúc tối ưu hiệu suất tính toán của CPU do một chương trình chiếm một không gian nhỏ hơn. Do đó chạy một chương trình không hoàn toàn nạp vào bộ nhớ vật lý mang đến lợi ích cho cả hệ thống lẫn người dùng.

**Bộ nhớ ảo:** Cung cấp một không gian luận lý cực kỳ lớn so với không gian vật lý khiến cho công việc của lập trình viên đơn giản hơn do họ có thể tập trung để xử lý vấn đề khác của chương trình mà không lo về số lượng vùng nhớ vật lý.

**Không gian địa chỉ ảo:** Một quá trình bắt đầu tại địa chỉ 0 và liên tục liền kề nhau như hình 1, tuy nhiên trên thực tế thì bộ nhớ vật lý thường được chia thành nhiều trang và thường gán cho quá trình những không gian không liên tục tùy thuộc vào bộ quản lý vùng nhớ(MMU).



Vùng nhớ ảo lớn hơn rất nhiều so với vùng nhớ vật lý

Bộ nhớ ảo cho phép những quá trình có thể chia sẻ tệp tin cũng như các thư viện với nhau dẫn đến một số lợi ích sau:

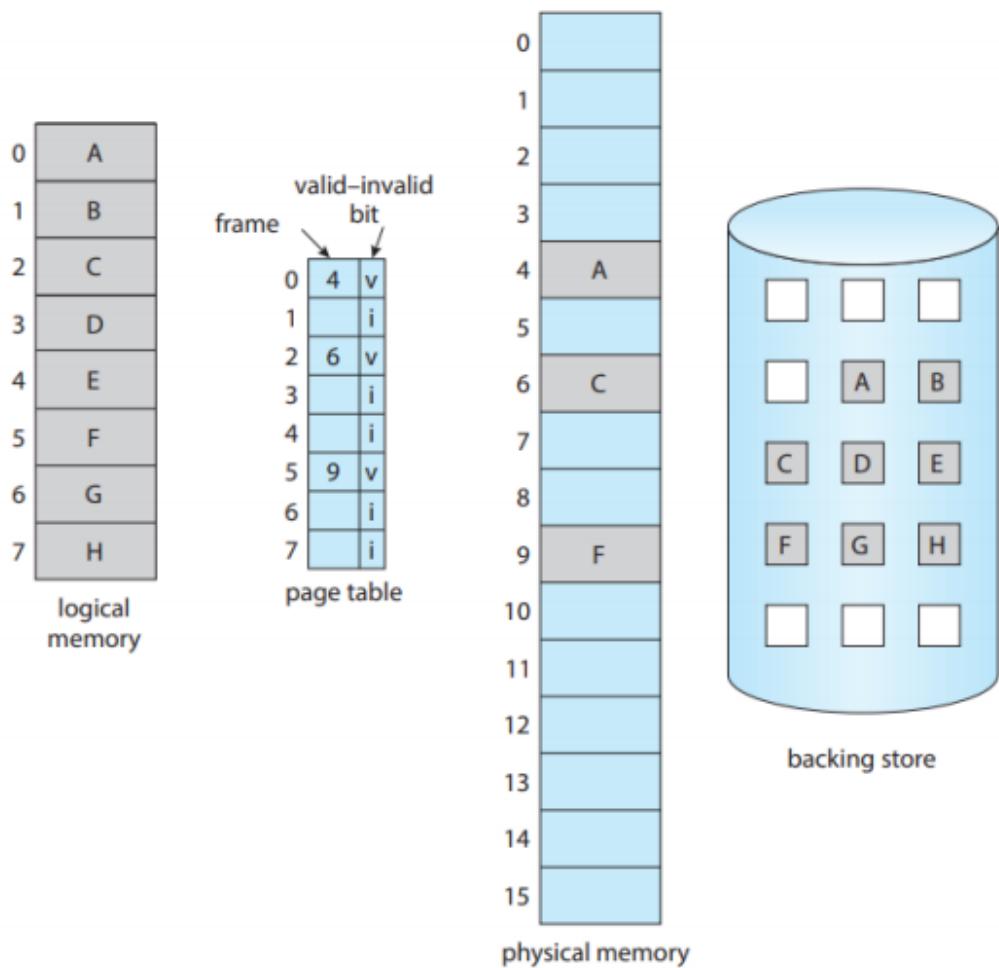
- \* Các thư viện hệ thống như Standard C có thể được chia sẻ với nhau bằng cách ánh xạ thành một địa chỉ ảo giữa tất cả các quá trình.
- \* Và tương tự, vùng nhớ giữa các quá trình cũng có thể chia sẻ với nhau.

## 2. Phân trang theo nhu cầu:

### 2.1 Khái niệm cơ bản:

Một trang chỉ được tải lên bộ nhớ khi thực sự cần thiết dẫn đến khi một chương trình đang thực thi sẽ có một số trang nằm trong bộ nhớ chính một số nằm trong bộ nhớ

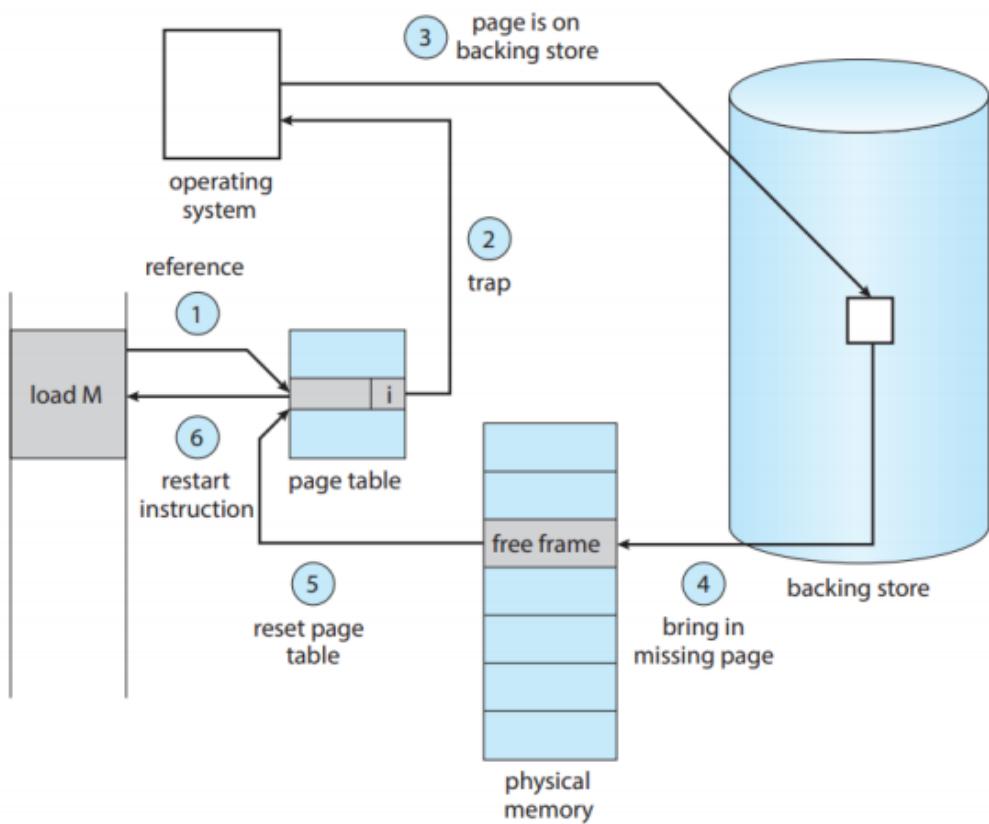
thứ cấp. Sơ đồ bit hợp lệ - không hợp lệ ở hình 2 dùng cho mục đích phân biệt 2 bộ nhớ này.



Một vài trang không nằm trong bộ nhớ chính

Khi bit được chỉnh thành valid nghĩa là trong bộ nhớ chính đã tồn tại trang đó và ngược lại không tồn tại hoặc tồn tại trong bộ nhớ thứ cấp sẽ là invalid. Khi truy cập vào trang được đánh dấu là invalid sẽ gây ra lỗi [page fault]. Lúc đó phần cứng cho việc phân trang sẽ gây ra một cái bẫy cho hệ điều hành và là kết quả khi hệ điều hành không thể đưa trang chỉ định vào bộ nhớ chính. Thủ tục xử lý page fault qua các bước:

1. Kiểm tra tham chiếu này hợp lệ hay không hợp lệ.
2. Nếu tham chiếu mà không hợp lệ, kết thúc quá trình và ngược lại nếu là hợp lệ nhưng vẫn chưa đưa vào trang chúng ta sẽ đưa nó vào.
3. Tìm một khung trống (bằng cách lấy từ danh sách các khung trống).



#### Các bước xử lý lỗi trang

4. Định thời một toán tử cho bộ nhớ thứ cấp để đọc trang được chỉ định vào trong khung trống mới được cấp phát.
5. Khi bộ nhớ thứ cấp hoàn thành công việc sẽ sửa đổi bảng nội của quá trình báo hiệu rằng trang đó đã nằm trong bộ nhớ chính.
6. Khởi động lại lệnh đã bị dừng bởi bẫy.

Trong trường hợp tệ nhất khi thực thi một quá trình mà không có trang nào trong bộ nhớ sẽ xảy ra page fault và sau đó trang này được chuyển vào bộ nhớ tiếp tục như vậy cho đến khi tất cả mọi trang cần thiết nằm trong bộ nhớ chính. Một yêu cầu quan trọng đối với phân trang theo yêu cầu là khả năng khởi động lại lệnh sau khi xảy ra page fault. Nếu page fault xảy ra trong quá trình nạp lệnh chúng ta có thể khởi động lại và nạp lệnh lại lần nữa, nếu xảy ra trong quá trình thực thi tính toán chúng ta phải nạp toán tử, giải mã lệnh lại và nạp lệnh lại một lần nữa.

## 2.2 Danh sách khung trống:

Khi page fault xảy ra, hệ điều hành phải chuyển trang chỉ định từ bộ nhớ thứ cấp sang bộ nhớ chính, hầu hết các hệ điều hành đều duy trì một danh sách khung trống để đáp ứng yêu cầu này. Thông thường hệ điều hành phân bổ khung trống theo kỹ thuật zero-fill-on-demand nội dung của khung trống sẽ được zeroed-out trước khi phân bổ. Khi một hệ thống khởi động, tất cả bộ nhớ khả dụng sẽ được đặt trong danh sách freeframe tất cả bộ nhớ khả dụng sẽ được đặt trong danh sách khung trống.

## 2.3 Hiệu suất của phân trang theo nhu cầu:

Kỹ thuật phân trang theo nhu cầu có ảnh hưởng rất lớn đến hiệu suất của hệ thống máy tính. Để làm rõ điều này hãy tính toán thời gian truy cập hiệu quả cho trang nhớ theo nhu cầu. Giả thiết rằng thời gian truy cập vùng nhớ ký hiệu ma là 10ns. Nếu không có page fault xảy ra thời gian truy cập hiệu quả bằng với thời gian truy cập vùng nhớ. Nếu có page fault xảy ra chúng ta đọc trang được chỉ định trong bộ nhớ thứ cấp và sau đó truy cập đến nó. Đặt p là xác suất xảy ra page fault, thời gian truy cập hiệu quả sẽ là:

$$\text{effective access time} = (1-p) * \text{ma} + p * \text{page fault time}.$$

Trong mọi trường hợp, có 3 thành phần chính ảnh hưởng đến thời gian xử lý page fault:

- \* Thời gian cho việc interrupt khi có page fault.
- \* Thời gian đọc một trang trong bộ nhớ thứ cấp.
- \* Thời gian khởi động lại quá trình.

Yếu tố đầu và cuối có thể giảm lược bằng cách viết mã thật cẩn thận. Giả sử chúng ta sử dụng HDD với thời gian chuyển trang là 8ms (độ trễ trung bình của đĩa cứng là 3ms thời gian tìm kiếm là 5ms thời gian di chuyển là 0.05ms). Với thời gian truy cập là 200ns, thời gian truy cập hiệu quả sẽ là:

$$\text{effective access time} = (1 - p) * 200 + p * 8000000 = 200 + 7,999,800 * p.$$

Từ kết quả trên cho thấy thời gian truy cập hiệu quả bị ảnh hưởng rất nhiều bởi xác suất  $p$ . Nếu một truy cập vượt quá 1000 lần page fault máy tính sẽ bị chậm lại với hệ số 40. Nếu chúng ta muốn hiệu suất kém hơn hiệu suất lý tưởng 10 phần trăm thì phải giữ xác suất ở mức  $p < 0.0000025$ .

### 3. Copy-on-write:

Tạo quá trình thông qua lời gọi hệ thống fork() có thể sử dụng kỹ thuật tương tự như việc chia sẻ trang, kỹ thuật này giúp tăng tốc độ cũng như giảm thiểu số trang cần khởi tạo cho một quá trình mới. Lời gọi hệ thống fork() tạo một quá trình con bằng cách sao chép cha của nó, nghĩa là hoạt động dựa trên nguyên lý sao chép không gian địa chỉ dành cho con và các trang của quá trình cha. Thực tế các quá trình con có thể sử dụng lời gọi hệ thống exec() ngay lập tức sau khi được tạo ra nên việc copy không gian địa chỉ của cha là không cần thiết. Thay vào đó chúng ta sẽ sử dụng kỹ thuật copy-on-write cho phép quá trình cha và con chia sẻ các trang với nhau. Mỗi khi một quá trình ghi vào trang được đánh dấu là copy-on-write một bản sao chép của nó sẽ được tạo ra.

Ví dụ như một quá trình có ý định sửa chữa một trang chứa các ch่อง với các trang được thiết lập là copy-on-write. Hệ điều hành sẽ lấy một khung trong danh sách khung trống và tạo một bảng sao của trang này, ánh xạ nó vào không gian địa chỉ của quá trình con. Khi sử dụng kỹ thuật copy-on-write chỉ có các trang được sửa đổi

bởi một trong hai quá trình được sao chép, những trang không được sửa chữa sẽ được chia sẻ giữa quá trình cha và con (những trang chứa các đoạn mã thực thi). Kỹ thuật này được ứng dụng phổ biến trong các hệ điều hành như Linux, Windows, MacOS...

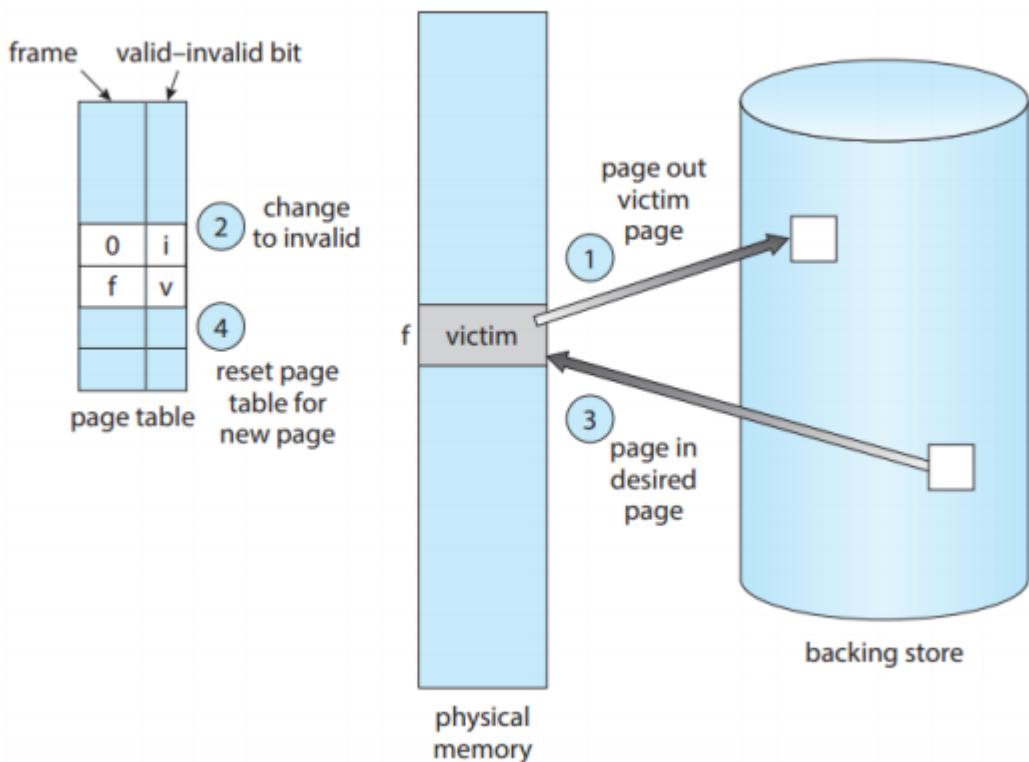
## 4. Thay trang

### 4.1 khái niệm thay thế trang:

Cách tiếp cận của kỹ thuật này là khi không còn khung trống sẽ tìm các khung mà gần đây không được sử dụng và giải phóng nó bằng cách ghi nội dung của nó đến không gian trao đổi đồng thời thay đổi bảng phân trang để chỉ ra rằng trang đó đã không còn nằm trong bộ nhớ chính. Các bước thực hiện như sau:

1. Tìm kiếm trong bộ nhớ thứ cấp trang cần thay.
2. Tìm một khung trống: Nếu không có khung trống, sẽ dụng giải thuật thay trang để lấy ra một khung trong bộ nhớ sao chép nó vào bộ nhớ thứ cấp đồng thời thay đổi bảng phân trang.
3. Đọc trang cần thay cho vào khung trống đó đồng thời thay đổi giá trị bảng phân trang.
4. Tiếp tục thực thi quá trình đã bị dừng do page fault.

Chú ý là nếu không còn khung trống chúng ta sẽ phải chuyển đổi 2 trang dẫn đến thời gian xử lý page fault tăng gấp đôi ảnh hưởng hiệu suất của hệ thống. Chúng ta có thể xử lý vấn đề này bằng cách sử dụng những bit sửa chữa (hay còn gọi là bit dơ). Bit này sẽ bị thay đổi khi mà trang tương ứng của nó trong bộ nhớ bị sửa chữa.



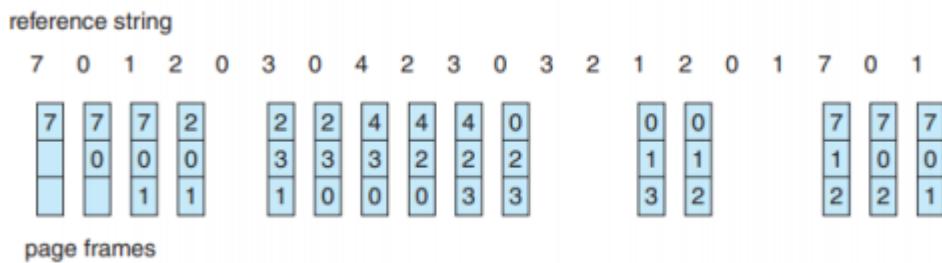
Minh họa cho việc thay trang

Chúng ta sẽ kiểm tra những bit này khi muốn tìm kiếm trang cần thay thế, nếu bit này là hợp lệ cho biết rằng đã được thay đổi từ khi được đọc vào từ bộ nhớ thứ cấp, trong trường hợp này chúng ta phải ghi nó lại vào bộ nhớ thứ cấp. Nếu bit này là không hợp lệ nghĩa là trang này chưa được thay đổi gì từ khi được đọc vào bộ nhớ chính do đó chúng ta không cần phải cập nhật nó lại trong bộ nhớ thứ cấp. Phương pháp này có thể giảm đáng kể thời gian xử lý lỗi trang do nó đã giảm một nửa thời gian I/O nếu trang chưa được thay đổi giá trị.

#### 4.2 Thay thế trang kiểu FIFO:

Giải thuật đơn giản nhất cho việc thay trang là first-in, first-out. Khi mà một trang phải bị thay thế thì hệ thống sẽ chọn trang cũ nhất (trang đầu tiên được nạp vào bộ nhớ). Chúng ta có thể tạo một hàng đợi chứa tất cả các trang trong bộ nhớ chính, khi cần

chỉ việc thay thế một trang ngay đầu hàng đợi đồng thời chèn một trang vào cuối hàng đợi.

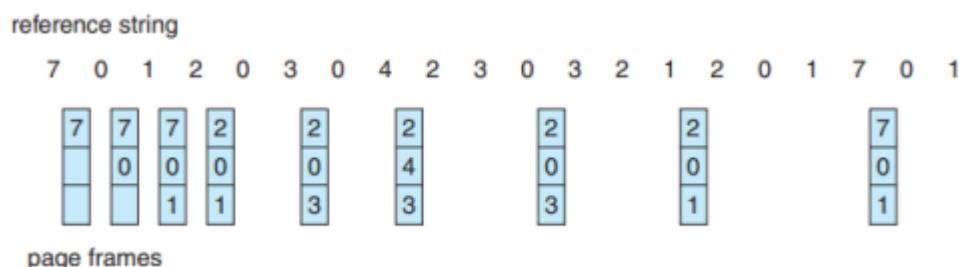


Ví dụ cho giải thuật FIFO

Ưu điểm của giải thuật này là mức độ tường minh cũng như có thể dễ dàng hiện thực song hiệu quả của nó không phải lúc nào cũng tốt. Chúng ta có thể tăng thêm hiệu suất của giải thuật này bằng cách khởi tạo nhiều khung cho bộ nhớ chính sẽ làm giảm khả năng xảy ra page fault nhưng điều này không phải lúc nào cũng đúng với bằng chứng là hiện tượng Belady's anomaly khi mà có 4 khung trống lại xảy ra page fault nhiều hơn lúc có 3 khung.

#### 4.3 Thay thế trang tối ưu:

Hệ quả của việc tìm ra hiện tượng Belady's anomaly là sự ra đời của giải thuật thay trang tối ưu - giải thuật có tỷ lệ page fault thấp nhất và không bao giờ lo lắng xảy ra Belady's anomaly. Phát biểu của giải thuật chỉ đơn giản là "Thay thế những trang sẽ không được sử dụng trong thời gian dài nhất".

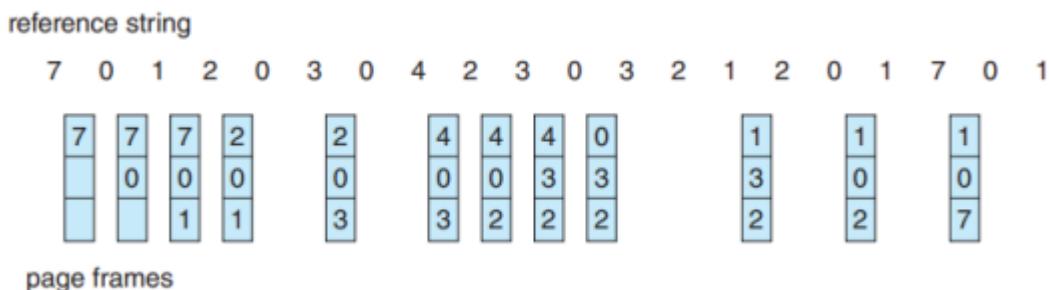


Thay trang kiểu tối ưu

Nhưng trên thực tế thì giải thuật này rất khó để hiện thực bởi vì chúng ta cần biết được những trang nào trong tương lai sẽ được (tương tự cho tình trạng của giải thuật SJF trong phần định thời CPU) sử dụng cho nên giải thuật này chỉ được áp dụng cho mục đích giáo dục so sánh với những giải thuật khác.

#### 4.4 Thay trang kiểu LRU:

Phương pháp LRU sẽ lưu lại lần sử dụng cuối cùng của trang tương ứng. Khi một trang cần được thay thế LRU sẽ chọn trang chưa được sử dụng trong một quãng thời gian dài nhất, có một sự tương đồng nhất định so với giải thuật tối ưu.



Thay trang kiểu LRU

Vấn đề là làm sao ta biết được số lần đã sử dụng của mỗi trang? Giải thuật này có thể cần sử hỗ trợ nhất định của các thiết bị phần cứng. Có 2 cách để giải quyết vấn đề này:

1. Counter trong trường hợp đơn giản nhất chúng ta liên kết với mỗi mục trong bảng phân trang một trường thời gian sử dụng và thêm vào trong clock của CPU hoặc là bộ đếm counter. Mỗi khi có tham chiếu đến trang sẽ tăng clock lên, bằng cách này sẽ có được số lần truy cập cuối cùng của mỗi trang.

2. Stack Một cách tiếp cận khác là tạo một cái chồng lưu giữ thứ tự của trang. Khi một trang được tham chiếu nó sẽ được đưa ra khỏi stack và đẩy lên đầu bằng cách này những trang thường xuyên sử dụng sẽ luôn ở trên đầu của chồng và những trang ít sử dụng sẽ ở đáy của chồng.

## 4.5 Thay trang xấp xỉ LRU

Không có nhiều hệ thống máy tính cung cấp phần cứng hỗ trợ cho phương pháp LRU chính thống. Trên thực tế nhiều hệ thống không cung cấp các hỗ trợ phần cứng. Một vài hệ thống cung cấp cho chúng ta những bit tham chiếu sẽ được thiết lập bởi phần cứng khi mà trang đó được tham chiếu. Vào thời điểm khởi tạo những bit này sẽ có giá trị 0 và sẽ được chỉnh thành 1 khi trang tương ứng được tham chiếu và cho chúng ta biết được là trang nào đã được sử dụng trang nào không nhưng sẽ không cung cấp cho ta thứ tự sử dụng của chúng. Thông tin này là nền tảng cho một vài giải thuật xấp xỉ LRU sau đây:

### 4.5.1 Additional-Reference-Bits

Chúng ta có thể đạt được thứ tự sử dụng mỗi trang bằng cách ghi lại những bit tham chiếu một cách đều đặn, sử dụng 8bit cho mỗi trang trong bảng phân trang. Khi trôi qua một khoảng thời gian (có thể là mỗi 100ms) hệ điều hành sẽ dịch những bit tham chiếu này đến vị trí cao hơn đồng thời loại bỏ bit ở vị trí thấp. Thanh ghi 8 bit này sẽ chứa đựng lịch sử truy cập của mỗi trang. Nếu một thanh ghi có giá trị 00000000 chứng tỏ đã không được truy cập trong khoảng 8 đơn vị thời gian, nếu một thanh ghi có giá trị 11111111 chứng tỏ ở mỗi đơn vị thời gian đã được tham chiếu ít nhất một lần. Và các thanh ghi này sẽ được so sánh với nhau thanh ghi có giá trị nhỏ nhất sẽ là LRU và trang tương ứng sẽ được thay ra ngoài.

### 4.5.2 Second-Chance

Nền tảng của giải thuật này là thay trang kiểu FIFO, khi một trang được chọn chúng ta sẽ quan sát bit tham chiếu của nó. Nếu bit có giá trị 0 chúng ta tiến hành thay trang, nếu bit là 1 chúng ta cho trang đó cơ hội thứ hai và chọn trang FIFO tiếp theo. Khi một trang được cho cơ hội thứ 2 bit tham chiếu sẽ bị xóa và thời gian sẽ được thiết lập bằng thời gian hiện tại. Như vậy một trang được cho cơ hội thứ 2 sẽ không

bao giờ bị thay thế cho đến khi những trang khác bị thay thế hoàn toàn ( hoặc trao cơ hội thứ 2 ).

Một cách để hiện thực giải thuật này là sử dụng một hàng đợi xoay vòng. Một con trỏ sẽ chỉ cho biết trang nào được thay thế tiếp theo, khi có nhu cầu thay trang con trỏ sẽ tìm kiếm trang có bit tham chiếu là 0 và xóa nó, thay thế trang cũ đồng thời chèn trang mới vào trong hàng đợi. Với trường hợp tệ nhất khi tất cả bit đều là 1 con trỏ phải duyệt qua toàn bộ hàng đợi và xóa bỏ bit tham chiếu của nó. Lúc này ta sẽ áp dụng giải thuật FIFO.

#### 4.5.3 Enhanced Second-Chance

Chúng ta có thể cải tiến giải thuật second-chance bằng cách thiết lập bit tham chiếu và bit sửa chữa thành một đôi có thứ tự. Với một đôi như vậy chúng ta sẽ có 4 trường hợp sau đây:

(0, 0) Gần đây trang này không được tham chiếu cũng như sửa chữa, là trang tốt nhất để thay thế.

(0, 1) Gần đây không được sử dụng nhưng đã có sự thay đổi, không quá tốt để thay thế vì phải ghi lại vào bộ nhớ thứ cấp.

(1, 0) Gần đây đã sử dụng nhưng chưa có thay đổi, có lẽ sẽ được sử dụng lại.

(1, 1) Gần đây đã sử dụng và sửa đổi, sớm thôi sẽ được sử dụng lại.

#### 4.6 Counting-Based

Có nhiều thuật toán khác có thể được sử dụng để thay thế trang. ví dụ, chúng ta có thể giữ một bộ đếm số lượng tham khảo đã được thực hiện cho mỗi trang với 2 chiến lược phát triển sau:

\* Thường xuyên sử dụng ít nhất (LFU) sẽ thay thế trang có số đếm nhỏ nhất. Lý do là trang được sử dụng thường xuyên sẽ có số đếm lớn nhưng có một vấn đề là một số trang sẽ được sử dụng thường xuyên trong một khoảng thời gian nhưng sau đó không cần thiết nữa. Để giải quyết vấn đề này là dịch biến đếm phải sau mỗi một khoảng thời gian nhất định.

\* Thường xuyên sử dụng nhiều nhất (MFU) dựa trên giả thiết là trang có biến đếm nhỏ nhất là trang vừa mới thêm vào bộ nhớ và có thể sẽ được sử dụng trong tương lai.

Như chúng ta có thể thấy, cả sự thay thế của MFU và LFU đều không phổ biến. Việc thực hiện các thuật toán này rất tốn kém và chúng không gần đúng với OPT.

#### 4.7 Page-Buffering

Hệ thống thường sẽ lưu giữ một nhóm các khung trống, khi lỗi trang xảy ra một khung sẽ được chọn như những giải thuật trên. Tuy nhiên trang chỉ định sẽ được đọc vào khung trống trước khi trang thay thế được ghi ra bộ nhớ thứ cấp. Thủ tục này cho phép quá trình có thể khởi động lại sớm nhất mà không cần phải chờ trang thay thế được ghi ra ngoài.

Ý tưởng này có thể được mở rộng bằng cách duy trì một danh sách các trang sửa đổi, khi mà thiết bị phân trang không hoạt động một trang sửa đổi sẽ được chọn và ghi vào bộ nhớ thứ cấp và reset lại bit sửa đổi. Cách làm này sẽ tăng xác xuất khi một trang được chọn để thay thế không cần phải ghi ra ngoài bộ nhớ thứ cấp.

#### 4.8 Ứng dụng và thay trang

Trong trường hợp nhất định khi các ứng dụng truy cập dữ liệu thông qua bộ nhớ ảo của hệ điều hành sẽ cho kết quả rất tệ nếu hệ điều hành không cung cấp buffering ví dụ như đối với những ứng dụng sử dụng bộ nhớ lớn như database sẽ có những trình quản lý bộ nhớ riêng sử dụng hiệu quả hơn so với hệ điều hành phát triển cho mục đích rộng rãi.

Trong một ví dụ khác, kho dữ liệu thường xuyên thực hiện lưu trữ tuần tự lớn, sau đó là tính toán và ghi. Giải thuật LRU sẽ bỏ những trang cũ và thay thế trang mới trong khi ứng dụng lại cần tác vụ đọc những trang cũ, lúc này MRU sẽ là một lựa chọn đúng đắn hơn so với LRU.

Hệ điều hành có thể cấp quyền truy cập trực tiếp vào đĩa nếu cung cấp chế độ raw disk bỏ qua những dịch vụ cho hệ thống tệp tin như là buffering, locking, space allocation....

## 5 Allocation of Frames

Vậy làm thế nào để ta phân bổ số lượng cố định các bộ nhớ trống cho các quá trình khác nhau? Nếu chúng ta có 93 khung trống và 2 quá trình thì mỗi quá trình sẽ lấy bao nhiêu khung?

### 5.1 Cấp phát số lượng khung nhỏ nhất

Mỗi quá trình sẽ cần số lượng nhỏ nhất các khung, tuy nhiên số lượng khung cung cấp cho mỗi quá trình giảm thì số lượng lỗi trang sẽ tăng lên ảnh hưởng rõ rệt đến hiệu suất thực thi của quá trình. Số lượng khung tối thiểu được xác định bởi kiến trúc máy tính ví dụ nếu lệnh move thuộc kiến trúc máy tính đó có nhiều hơn 1 word thì lệnh sẽ chiếm 2 khung, ngoài ra, nếu mỗi trong hai toán hạng của nó có thể là các tham chiếu gián tiếp sẽ cần tổng cộng 6 khung.

### 5.2 Giải thuật cấp phát khung

Cách đơn giản nhất để cấp phát cho n quá trình khi có m khung là chia đều, mỗi quá trình sẽ có m/n khung. Ví dụ như nếu có 93 khung và 5 quá trình thì mỗi quá trình sẽ có 18 khung. Cách này được gọi là phân bổ đều.

Tuy nhiên những quá trình khác nhau sẽ cần sử dụng số lượng vùng nhớ khác nhau. Ví dụ như một quá trình chỉ 10KB và một quá trình 100KB đều chạy đồng thời trên một hệ thống có 62 khung trống thì việc cấp phát cho mỗi quá trình là 31 khung trống có vẻ không hợp lý lầm nếu không muốn nói là quá lãng phí cho quá trình 10KB. Chúng ta có thể giải quyết vấn đề này bằng cách phân bổ theo tỷ lệ. Số lượng khung cấp phát cho mỗi quá trình sẽ là:

$$a = s/S * m$$

Với  $a$  là số lượng khung cho quá trình đó,  $s$  là kích thước bộ nhớ ảo cho quá trình đó,  $m$  là tổng số khung trống.

Như vậy chúng ta sẽ phân cho quá trình 10KB là 5 khung và quá trình 100KB là 56 khung bởi vì

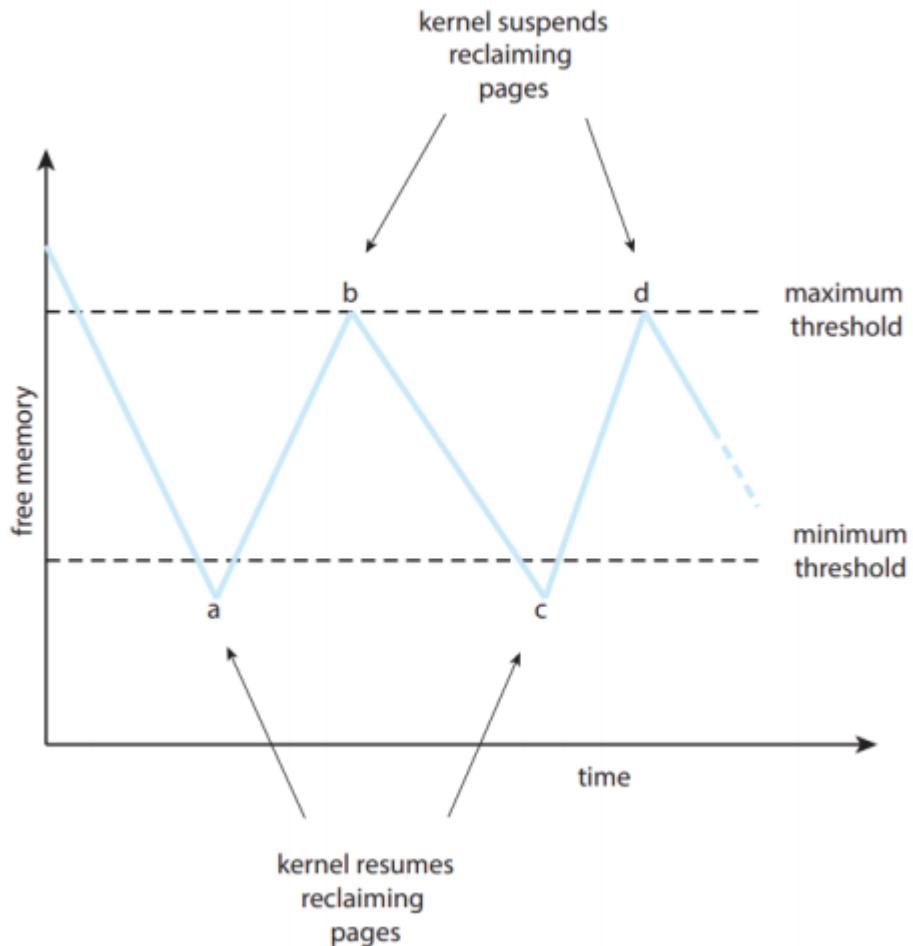
$$10/110 * 62 = 5$$

$$100/110 * 62 = 56$$

### 5.3 Phân bổ cục bộ vs toàn cục

Một yếu tố quan trọng khác trong cách các khung được phân bổ cho các quá trình khác nhau là việc thay trang. Chúng ta có thể phân loại các giải thuật thay trang thành 2 loại là thay toàn cục và thay cục bộ. Thay toàn cục cho phép quá trình lựa chọn một khung trong tập hợp toàn bộ các khung thậm chí kể cả khi khung đó đang được sử dụng bởi một quá trình khác trong khi thay cục bộ chỉ cho phép quá trình lựa chọn một trong các khung đã được phân bổ riêng cho nó mà thôi.

Để hiện thực chiến lược thay thế toàn cục với cách tiếp cận này, ta phải thỏa mãn tất cả các yêu sử dụng vùng nhớ từ danh sách khung trống nhưng thay vì đợi danh sách giảm xuống 0, ta kích hoạt thay thế trang khi danh sách nằm dưới một ngưỡng nhất định. Chiến lược này đảm bảo luôn có bộ nhớ trống để cấp cho những yêu cầu mới.



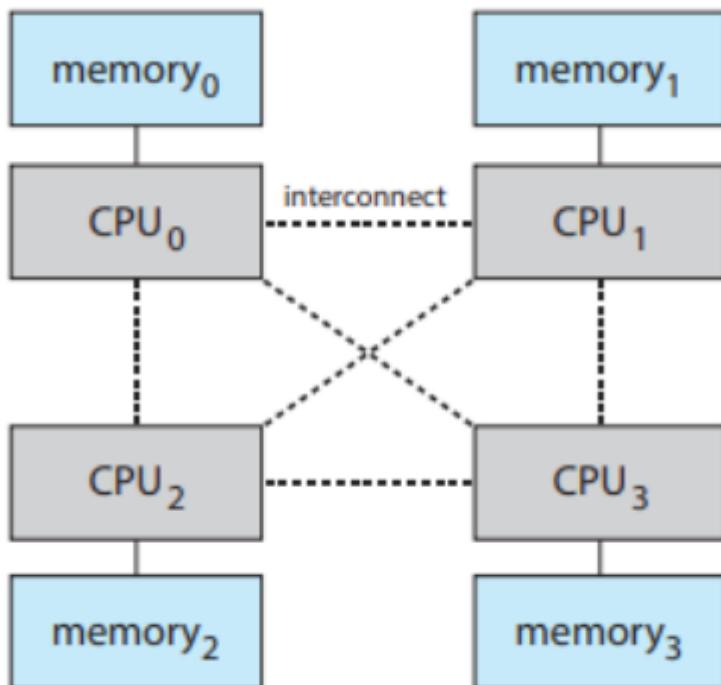
### Thu hồi trang

Mục đích của chiến lược là giữ dung lượng bộ nhớ trống trên ngưỡng tối thiểu. Khi vùng nhớ trống xuống dưới ngưỡng tối thiểu, kernel sẽ bắt đầu đi thu hồi các trang từ tất cả các quá trình trong hệ thống cho đến khi số lượng vùng nhớ trống nằm trên ngưỡng tối đa thì sẽ bị đình chỉ và chỉ được tiếp tục khi vùng nhớ trống đạt dưới ngưỡng tối thiểu một lần nữa. Quá trình này sẽ được tiếp diễn khi nào hệ thống còn hoạt động.

### 5.4 Non-Uniform Memory Access (NUMA)

Cho đến nay, tất cả những bộ nhớ chính được sản xuất đều ít nhất có tốc độ truy cập là ngang nhau. Tuy nhiên đối với hệ thống NUMA đa CPU thì một CPU có thể truy cập đến vài thành phần của bộ nhớ chính với tốc độ nhanh hơn phần còn lại và liên quan chặt chẽ đến các liên kết nội giữa CPU và bộ nhớ trong hệ thống. Mỗi CPU sẽ có bộ

nhớ cho riêng nó (hình 9), tốc độ truy xuất đến bộ nhớ riêng này sẽ nhanh hơn so với việc truy xuất đến bộ nhớ riêng của CPU khác. Hệ thống NUMA sẽ chậm hơn các hệ thống thông thường tuy nhiên nó có thể chứa nhiều CPU hơn nên có thể đạt được khả năng song song và thông lượng ở một đẳng cấp cao hơn.



Hệ thống NUMA

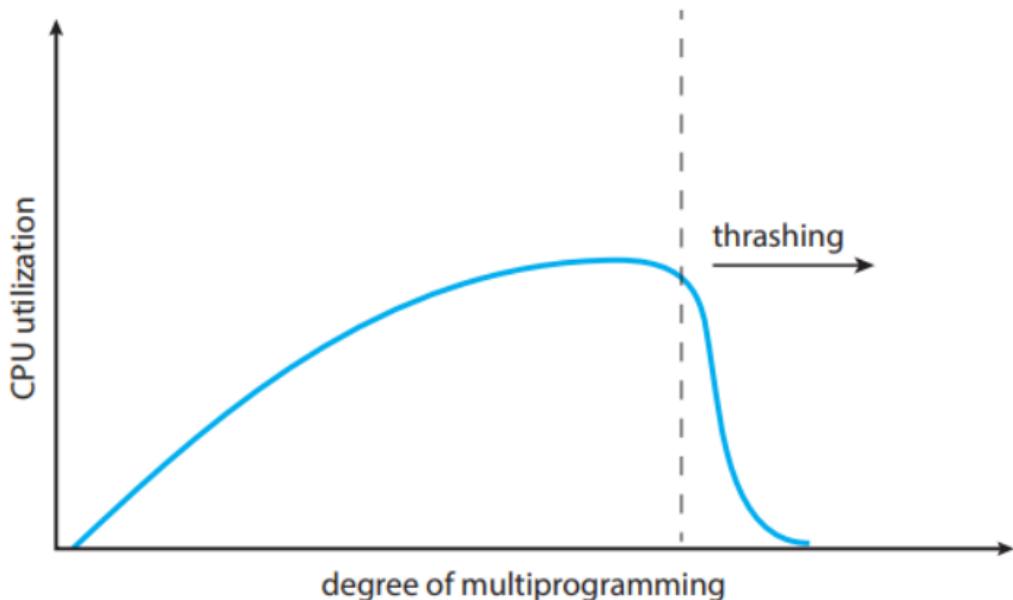
## 6 Thrashing

Phân trang hoạt động với cường độ cao sẽ được gọi là thrashing. Một quá trình bị thrashing khi nó dành nhiều thời gian để phân trang hơn là thực thi và hiện tượng thrashing này ảnh hưởng rất lớn đến hiệu suất của hệ thống.

### 6.1 Nguyên nhân

Hệ điều hành sẽ giám sát mức độ sử dụng CPU, nếu thông số này quá thấp ta tăng lên bằng cách thêm vào hệ thống quá trình mới. Khi quá trình mới cần thêm khung sẽ gây ra lỗi trang và lấy những khung của quá trình khác (do giải thuật thay trang toàn cục) và các quá trình đó cần khung trang sẽ tiếp tục lấy của quá trình khác và cứ thế tiếp tục. Khi một quá trình phải chờ đợi thay trang sẽ làm giảm mức độ sử dụng CPU của hệ thống.

Bộ định thời CPU thấy mức độ sử dụng CPU giảm sẽ tiếp tục thêm vào nhiều quá trình khác để tăng mức độ xử lý. Hệ quả là mức độ sử dụng CPU sẽ bị suy giảm thậm chí càng ngày càng nhiều dẫn đến sẽ không có công việc nào được hoàn thành vì toàn bộ thời gian đã được sử dụng cho việc thay trang.

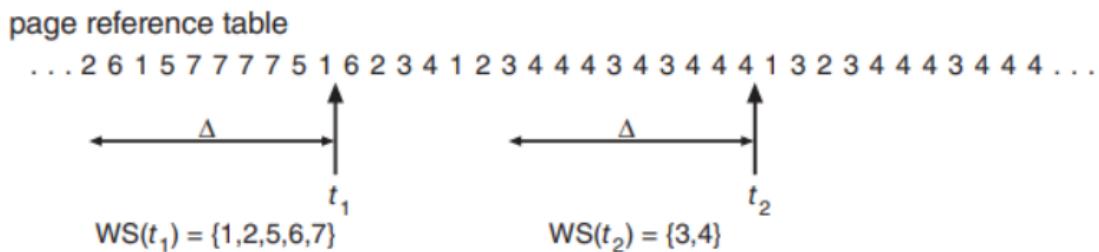


Quan hệ giữa thông số CPU và độ đa chương

Để ngăn ngừa thrashing xảy ra ta phải cung cấp cho mỗi quá trình đúng với số lượng khung mà nó cần bằng cách tiếp cận được gọi là Locality model. Một Locality là tập hợp các trang được sử dụng cùng nhau.

## 6.2 Mô hình working-set

Mô hình này dựa trên những giả định của locality, sử dụng một tham số là  $\Delta$  để định nghĩa một cửa sổ working-set với ý tưởng là xem xét những trang tham chiếu  $\Delta$  gần nhất. Tập hợp của các trang  $\Delta$  tham chiếu đến gần nhất được gọi là working set. Nếu một trang được sử dụng nhiều lần sẽ nằm trong working set và ngược lại sẽ nằm ngoài working set  $\Delta$  đơn vị thời gian so với lần tham chiếu cuối cùng của nó.



Mô hình working-set

Với ví dụ trên nếu  $\Delta = 10$  thì working set ở vị trí  $t_1$  là 1, 2, 5, 6, 7 ở thời điểm  $t_2$  chỉ còn 3, 4. Thuộc tính quan trọng nhất của working set là kích thước của nó. Nếu kích thước của một working set là  $WSS$  cho mỗi quá trình trong hệ thống thì  $D = WSS$  với  $D$  là tổng số khung cần thiết. Nếu một quá trình thường xuyên sử dụng trang trong working set của nó sẽ cần  $WSS$  khung. Nếu  $D > m$  sẽ xảy ra thrashing bởi vì một vài quá trình sẽ không có đủ khung cho nó.

Hệ điều hành sẽ theo dõi working set của mỗi quá trình và phân bổ đủ số lượng khung cần thiết với kích thước của working-set. Nếu còn lại đủ số khung thêm cho quá trình mới, quá trình đó sẽ được khởi tạo. Nếu tổng số kích thước working-set tăng vượt quá số lượng khung hệ điều hành sẽ lựa chọn và định chỉ một quá trình, quá trình này sẽ được khởi động lại sau.

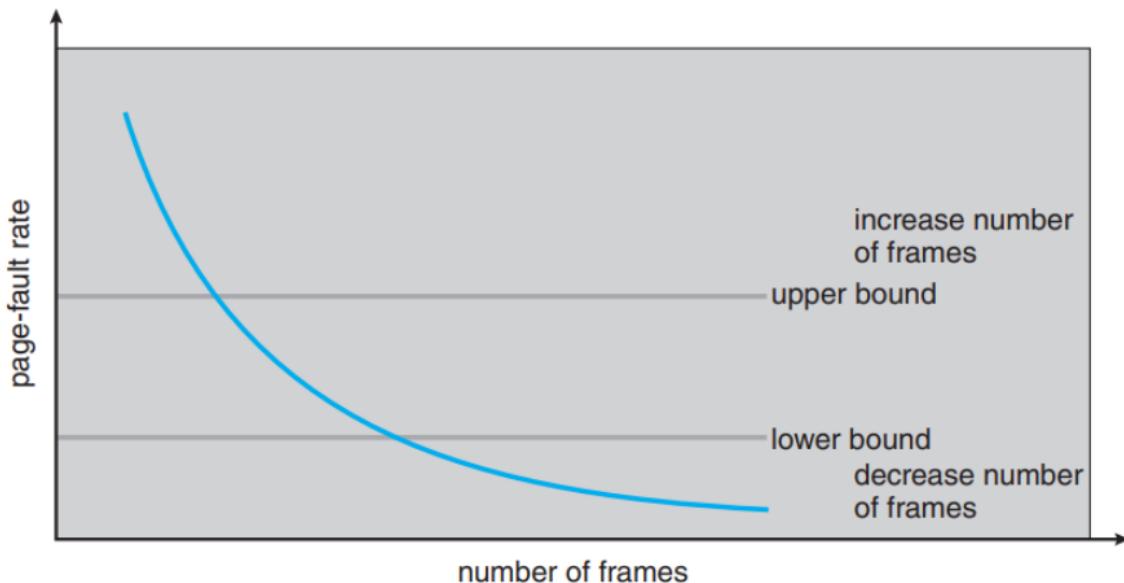
Ta có thể xấp xỉ mô hình working-set với bộ đếm thời gian ngắn và bit tham chiếu. Ví dụ như có  $\Delta$  là 10000 tham chiếu và bộ đếm sẽ ngắn mỗi khi có 5000 tham chiếu. Khi bị ngắn ta sao chép và loại bỏ bit tham chiếu của mỗi trang. Nếu xảy ra lỗi trang, ta kiểm tra bit tham chiếu hiện tại và 2 bit trong bộ nhớ để xác

định trang đó có được sử dụng trong khoảng 10000 đến 15000 tham chiếu cuối cùng. Nếu được sử dụng sẽ có ít nhất một bit được bật lên và trang này sẽ nằm trong working set.

### 6.3 Tần suất lỗi trang

Mô hình working-set chỉ hữu dụng khi được sử dụng cho công cuộc chuẩn bị song không phải một cách hiệu quả để kiểm soát thrashing. Một chiến lược có cách tiếp cận trực tiếp vấn đề này là tần suất lỗi trang (PFF).

Thrashing có tỷ lệ xảy ra lỗi trang cao, do đó để ngăn ngừa thrashing ta phải kiểm soát được tỷ lệ lỗi trang. Khi tỷ lệ này quá cao ta biết được quá trình cần thêm nhiều khung ngược lại nếu tỷ lệ quá thấp nghĩa là một quá trình dư thừa quá nhiều khung. Chúng ta có thể thiết lập chặn trên và dưới cho tỷ lệ lỗi trang mong muốn như hình 12.



Quan hệ giữa tần suất lỗi trang và khung sử dụng

Nếu tần suất sai trang vượt quá chặn trên ta sẽ cấp phát thêm khung cho quá trình và ngược lại lấy lại khung của quá trình nếu tần suất thấp hơn chặn dưới. Do

đó ta có thể kiểm soát tần suất sai trang bằng những phép đo trực tiếp để ngăn ngừa thrashing.

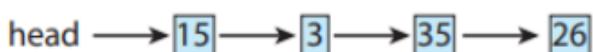
## 7 Nén bộ nhớ

Xem xét ví dụ dưới đây khi mà danh sách khung trống chứa 6 khung:

### free-frame list



### modified frame list



Danh sách trang trước khi nén

Giả sử rằng số khung hình miễn phí này nằm dưới một ngưỡng nhất định do đó kích hoạt thay thế trang. Thuật toán thay thế (giả sử ta sử dụng giải thuật xấp xỉ cho LRU) chọn bốn khung - 15, 3, 35 và 26 - để đặt vào danh sách khung miễn phí. Đầu tiên, ta đặt các khung này vào danh sách khung đã sửa đổi. Thông thường thì danh sách khung đã sửa đổi tiếp theo sẽ được ghi vào không gian hoán đổi, một cách khác là nén số khung có thể là 3 và lưu những khung đã nén này vào trong một danh sách khung duy nhất, cho phép hệ thống giảm sử dụng bộ nhớ mà không dùng đến các trang hoán đổi.

### free-frame list



### modified frame list



### compressed frame list



Danh sách trang sau khi nén

## 8 Phân bổ vùng nhớ kernel

Bộ nhớ kernel thường được phân bổ từ nhóm bộ nhớ trống khác với danh sách được sử dụng để đáp ứng các quá trình của người dùng thông thường. Có 2 lý do chính cho việc này.

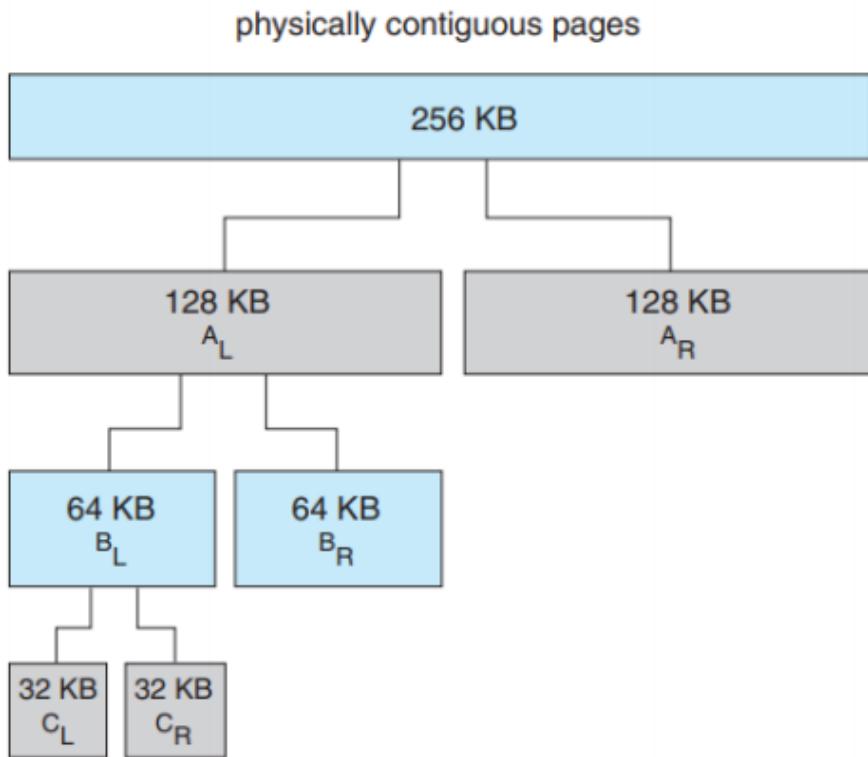
1\* Kernel yêu cầu bộ nhớ cho các cấu trúc dữ liệu có kích thước khác nhau, một số có kích thước nhỏ hơn một trang. Kết quả là kernel phải cố gắng giảm thiểu sự lãng phí do hiện tượng phân mảnh.

2\* Các trang được phân bổ cho các quá trình ở chế độ người dùng không nhất thiết phải liên tục trong bộ nhớ vật lý. Tuy nhiên một số thiết bị phần cứng tương tác trực tiếp với bộ nhớ vật lý có thể yêu cầu cấp phát vùng nhớ liên tục.

### 8.1 Hệ thống Buddy

Phân bổ bộ nhớ từ phân đoạn kích thước cố định bao gồm các trang liên tục trong bộ nhớ vật lý, kích thước phân đoạn này thường là một số lũy thừa của 2. Một yêu cầu phân bổ có kích thước không phải là lũy thừa của 2 sẽ được làm tròn lên số lũy thừa 2 gần nhất. Ví dụ yêu cầu 11KB thì sẽ cấp phát 16KB.

Giả sử một phân đoạn của bộ nhớ được khởi tạo 256KB và kernel yêu cầu 21KB. Phân đoạn này sẽ được chia làm 2 buddies mỗi cái sẽ có 128KB. Một trong 2 cái buddies đó sẽ tiếp tục chia làm 2 phần mỗi phần gồm 64KB và tiếp tục chia làm 2 phần mỗi phần 32KB. Lúc đó buddy có kích thước 32KB sẽ được cấp phát cho yêu cầu 21KB này.



Minh họa cho hệ thống buddy

Điểm mạnh của chiến lược này là có thể kết hợp các phần không sử dụng thành phần lớn hơn ví dụ kernel giải phóng đơn vị CL hệ thống có thể kết hợp CL và CR thành một phân đoạn 64 KB và phân đoạn BL này có thể kết hợp với BR để tạo thành phân đoạn 128KB và cuối cùng kết thúc với phân đoạn 256KB gốc. Tuy nhiên chiến lược này sẽ xảy ra hiện tượng phân mảnh khi một yêu cầu cấp phát 33KB ta chỉ có thể cấp phát cho nó 64KB là kích thước nhỏ nhất.

## 8.2 Phân bổ Slab

Chiến lược thứ 2 để phân bổ cho kernel là phân bổ kiểu slab, một cái slab sẽ được tạo thành từ 1 hoặc nhiều hơn các trang nhớ vật lý liên tiếp. Một cái cache sẽ bao gồm 1 hoặc nhiều slab, mỗi cache được dùng cho một cấu trúc dữ liệu độc nhất của kernel. Ví dụ một cache sẽ được dùng cho semaphore, một cách được dùng cho mô tả quá trình....

Giải thuật này sử dụng cache để lưu giữ một đối tượng của kernel. Khi một cache được tạo ra một số lượng object sẽ được đánh dấu là free được phân bổ vào cache. Khi cần một đối tượng mới cho cấu trúc dữ liệu kernel, bộ cấp phát có thể chỉ định bất kỳ đối tượng free nào từ cache để đáp ứng yêu cầu. Các đối tượng này sẽ được đánh dấu là used. Một slab sẽ tồn tại 3 trạng thái:

- 1 Full. Tất cả đối tượng trong slab đều được đánh dấu là used.
- 2 Empty. Tất cả đối tượng đều được đánh dấu là free.
- 3 Partial. Trong slab tồn tại cả 2 đối tượng free và used.

Bộ cấp phát đầu tiên sẽ cố gắng sử dụng đối tượng free trong slab partial. Nếu không thỏa mãn sẽ tìm đối tượng trong slab free và nếu không có slab empty ta sẽ tạo một slab mới và gán nó vào cache đồng thời sử dụng đối tượng này cho yêu cầu cấp phát.

## 9       Những thứ liên quan khác

### 9.1 Chuẩn bị phân trang

Đây là công đoạn nhằm mục đích giảm số lượng lõi trang xảy ra khi một quá trình được khởi tạo. Ta chuẩn bị toàn bộ hoặc một vài trang mà quá trình có lẽ sẽ cần trước khi nó được tham chiếu. Nhưng nếu những trang này không được sử dụng thì những thao tác I/O và vùng nhớ sẽ bị lãng phí.

Giả sử rằng có  $s$  trang được chuẩn bị trước và một phần nhỏ  $\alpha$  của các trang này được sử dụng ( $0 \leq \alpha \leq 1$ ). Liệu chi phí cho việc chuẩn bị trang trước  $s * \alpha$  có tốt hơn chi phí khi trang chuẩn bị không được sử dụng  $s * (1 - \alpha)$ . Nếu  $\alpha$  gần bằng 0 nghĩa là việc chuẩn bị này vô nghĩa ngược lại là có ích.

### 9.2 Kích thước trang

Vậy làm thế nào để chúng ta chọn một kích thước trang?. Với một không gian bộ nhớ ảo nhất định, giảm kích thước trang sẽ tăng số lượng các trang và do đó tăng kích thước của bảng trang. ví dụ: Đối với bộ nhớ ảo có 4MB sẽ có 4.096 trang trong số 1.024 byte nhưng chỉ có 512 trang 8.192 byte.

Câu trả lời là không hề có một phương pháp chọn kích thước tối ưu nào cả. Như chúng ta đã thấy, một số yếu tố (phân mảnh nội bộ, locality) sẽ tốt hơn đối với trang có kích thước nhỏ, trong khi những vấn đề còn lại (kích thước bảng, thời gian I / O) sẽ tốt hơn đối với trang có kích thước lớn. Tuy nhiên, xu hướng lịch sử là hướng tới kích thước trang lớn hơn, thậm chí cho các hệ thống di động. Phiên bản đầu tiên hệ điều hành (1983) đã sử dụng 4.096 byte làm giới hạn trên của kích thước trang và giá trị này là kích thước trang phổ biến vào năm 1990. Các hệ thống hiện đại có thể sử dụng trang với kích thước lớn hơn nhiều.

### 9.3 TLB Reach

TLB Reach là số lượng vùng nhớ có thể truy cập từ TLB và được tính bằng công thức  $TLB\ Reach = (TLB\ Size) * (Page\ Size)$ . Lý tưởng nhất thì working set của một quá trình sẽ được lưu trữ trong TLB nếu không quá trình sẽ sử dụng một lượng đáng kể thời gian giải quyết các tham chiếu bộ nhớ trong bảng phân trang.

Một cách tiếp cận khác để tăng TLB Reach là tăng kích cỡ trang hoặc cung cấp trang với nhiều kích cỡ. Tuy nhiên điều này dẫn đến tăng khả năng phân mảnh khi một số ứng dụng không yêu cầu kích thước trang lớn như vậy.

Nhiều kiến trúc cung cấp nhiều hơn một kích thước trang và hệ điều hành có thể được cấu hình để tận dụng lợi thế này. Ví dụ kích thước trang mặc định của Linux là 4KB nhưng cũng cung cấp cho người dùng những trang có kích thước lớn (2MB).

### 9.4 I/O Interlock

I/O Interlock Đôi khi một vài trang sẽ bị khóa trong bộ nhớ.

Consider I/O Các trang được sử dụng để sao chép tệp từ thiết bị phải bị khóa để phòng trường hợp bị thay ra bởi giải thuật thay trang.

Sử dụng bit lock có thể nguy hiểm: bit lock có thể được bật nhưng không bao giờ tắt nếu tình huống này xảy ra (vì một lỗi trong hoạt động hệ thống, ví dụ), khung bị khóa trở nên không sử dụng được.

## 9.5 Cấu trúc chương trình

Người dùng có thể cải thiện hiệu suất của hệ thống khi họ có một nhận thức cơ bản về cơ chế phân trang theo nhu cầu. Xét một ví dụ sau, giả định rằng mỗi trang có kích thước 128 word với đoạn mã sau đây, chúng ta chú ý rằng mỗi hàng sẽ được lưu trong 1 trang:

```
int i, j;
int[128][128] data;

for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i][j] = 0;
```

```
int i, j;
int[128][128] data;

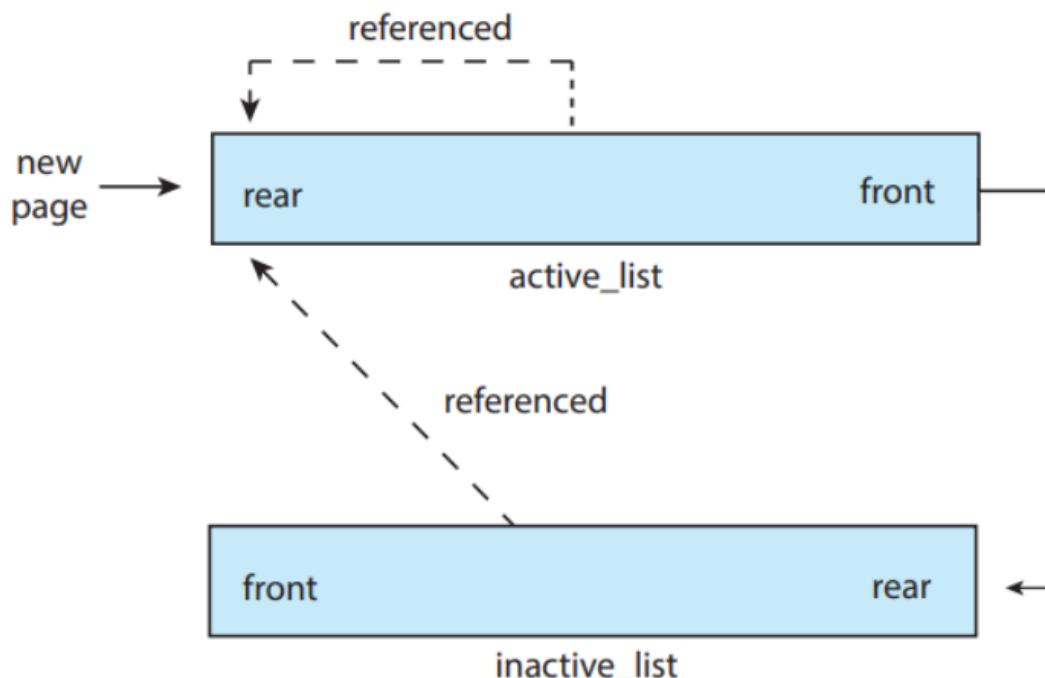
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i][j] = 0;
```

128 x 128 lõi trang (trái) và 128 lõi trang (phải)

## 10 Ví dụ thực tiễn

### 10.1 Linux

Linux sử dụng cơ chế phân trang theo yêu cầu, phân bổ các trang từ một danh sách các khung trống, sử dụng chính sách thay thế trang toàn cục tương tự như giải thuật xấp xỉ LRU. Để quản lý vùng nhớ Linux duy trì hai loại danh sách trang: danh sách hoạt động và danh sách không hoạt động.



Cấu trúc danh sách hoạt động và không hoạt động

Danh sách hoạt động chứa những trang gần đây được sử dụng ngược lại với danh sách không hoạt động. Mỗi trang có một bit truy cập được đặt bất cứ khi nào trang được tham chiếu. Khi một trang được tham chiếu bit truy cập sẽ được bật lên và di chuyển đến phần phía sau của danh sách hoạt động. Theo thời gian những trang ít được sử dụng nhất sẽ được đẩy lên đầu của danh sách và từ đó có thể được di chuyển đến phần phía sau của danh sách không hoạt động. Khi một trang của danh sách không hoạt động được tham chiếu thì cũng được đẩy vào phần phía sau của danh sách hoạt động.

## 10.2 Windows

Win 10 hiện thực bộ nhớ ảo sử dụng kỹ thuật phân trang theo yêu cầu bằng clustering giúp ta nhận diện locality của các tham chiếu bộ nhớ. Kích cỡ của một cluster phụ thuộc vào kiểu trang.

Khi một quá trình được tạo ra sẽ được gán cho một working-set tối thiểu với 50 trang và working-set tối đa với 345 trang.

- \* working-set tối thiểu là số lượng trang tối thiểu mà bộ nhớ đệm bảo có cho quá trình.
- \* working-set tối đa một quá trình có thể được cấp phát nhiều trang đến mức working-set tối đa.

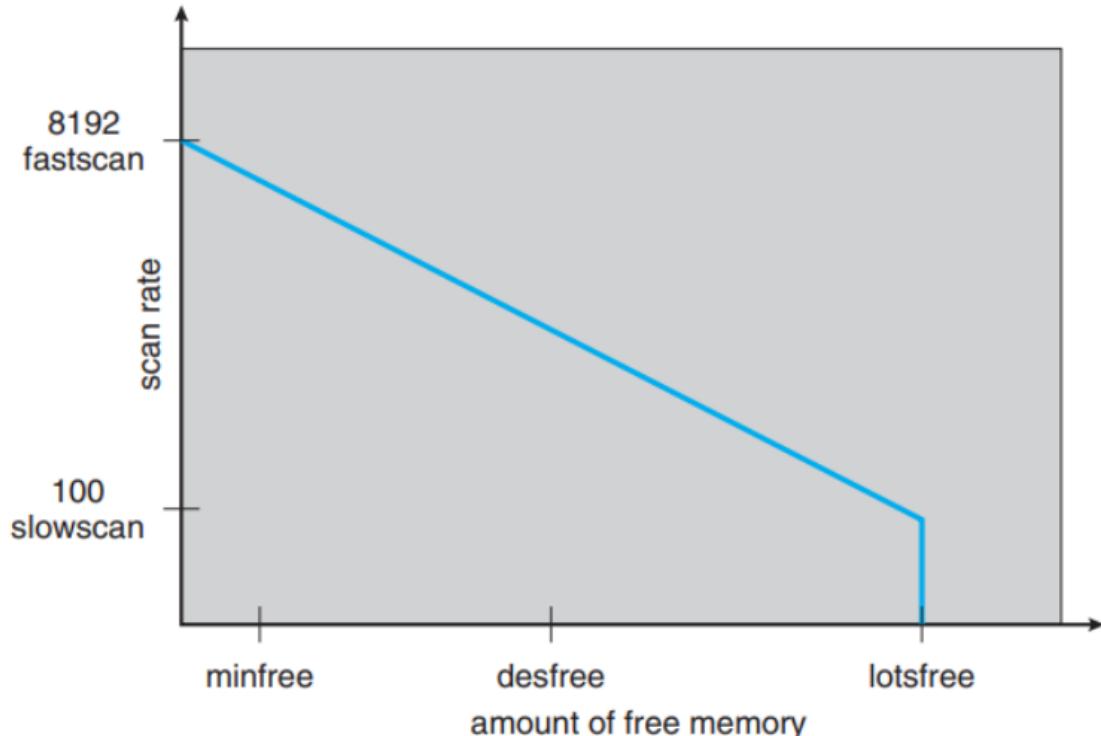
Windows sử dụng giải thuật xấp xỉ LRU kết hợp với những chiến lược thay thế toàn cục và cục bộ. Trình quản lý bộ nhớ ảo sẽ duy trì một danh sách các khung trống. Nếu có lỗi trang xảy ra mà working-set vẫn dưới mức tối đa trình quản lý sẽ cấp thêm khung cho quá trình đó. Tuy nhiên nếu không còn đủ khung trống thì kernel phải chọn và thu hồi trang từ working set của các quá trình bằng cách thay trang LRU cục bộ.

Khi số lượng khung trống lao xuống dưới ngưỡng tối thiểu trình quản lý sẽ sử dụng thay thế toàn cục để được biết đến là automatic working set trimming để thu hồi khung trống lại ở mức trên ngưỡng tối thiểu. Nếu một quá trình được cấp phát nhiều trang hơn working-set tối thiểu của nó sẽ bị trình quản lý thu hồi lại.

### 10.3 Solaris

Kernel sẽ duy trì một danh sách các trang trống tương ứng là tham số lotsfree thể hiện một ngưỡng để bắt đầu thay trang và thường được thiết lập với kích cỡ 1/64 của bộ nhớ vật lý. Kernel sẽ kiểm tra 4 lần mỗi giây để xem lượng khung trống có ít hơn lotsfree hay không. Nếu số trang trống nằm dưới ngưỡng lotsfree sẽ thực thi một quá trình có tên là pageout.

Giải thuật pageout dùng những tham số để kiểm soát tỷ lệ được biết đến là scanrate, đơn vị là số trang trên một giây và dao động từ slowscan đến fastscan.



Page scanner của Solaris

Nếu kích cỡ bị vùng nhớ còn trống đạt dưới giá trị của desfree, pageout sẽ chạy hàng trăm lần mỗi giây để giữ cho tối thiểu của desfree vùng nhớ còn trống. Nếu kernel không thể giữ cho vùng nhớ trống ở mức desfree trong khoảng 30 giây kernel sẽ bắt đầu hoán đổi các quá trình giải phóng tất cả các trang đã cấp phát cho quá trình bị hoán đổi.

Page-scanning có thể phân biệt được trang được cấp phát cho quá trình và trang được cấp phát cho tệp tin dữ liệu thông thường. Điều này được biết đến như là phân trang có sự ưu tiên hay priority paging.

## 11      Tổng kết

- Bộ nhớ ảo trừu tượng hóa bộ nhớ vật lý thành một mảng lưu trữ cực lớn.

- Lợi ích khi sử dụng bộ nhớ ảo bao gồm: (1) chương trình có thể lớn hơn bộ nhớ vật lý, (2) chương trình không cần hoàn toàn nạp vào bộ nhớ vật lý, (3) các quá trình có thể chia sẻ vùng nhớ, (4) quá trình có thể được khởi tạo hiệu quả hơn.
- Các trang chỉ được nạp vào bộ nhớ chính khi nó có nhu cầu được sử dụng trong quá trình thực thi (demand paging).
- Lỗi trang xảy ra khi truy cập một trang không nằm trong bộ nhớ chính. Trang đó phải được nạp lại từ bộ nhớ thứ cấp.
- Copy-on-write cho phép quá trình con chia sẻ không gian địa chỉ với quá trình cha.

Khi không còn số lượng khung trống, giải thuật thay trang sẽ chạy nhằm thay thế những trang mới vào vị trí trang cũ trong bộ nhớ chính.

- Thay toàn cục sẽ lựa chọn trang từ tất cả các quá trình trong khi thay cục bộ chỉ lựa chọn những trang được cấp phát cho nó.
- Thrashing xảy ra khi hệ thống dành nhiều thời gian cho việc thay trang hơn là thực thi.
- Một locality đại diện cho tập hợp những trang được sử dụng thường xuyên. Working set có nền tảng là locality và được định nghĩa là tập hợp các trang đang được sử dụng của quá trình.
- Nén vùng nhớ là kỹ thuật để nén một số lượng trang vào trong một trang duy nhất và được sử dụng như một biện pháp thay thế cho việc thay trang và ứng dụng cho các thiết bị di động không hỗ trợ thay trang.
- Kernel được phân bổ theo những vùng nhớ liên tục có kích thước khác nhau. 2 kỹ thuật chính là (1) buddy system và (2) slab allocation.
- TLB Reach đề cập đến dung lượng bộ nhớ có thể truy cập từ TLB và bằng với số lượng TLB \* kích thước trang.
- Linux, Windows và Solaris quản lý bộ nhớ ảo tương tự, sử dụng demand paging và copy-on-write. Mỗi hệ thống cũng sử dụng một biến thể của xấp xỉ LRU được gọi là thuật toán clock.

## 12 Câu hỏi

Giả sử ta có 3 khung trống. Cho chuỗi tham chiếu sau đây:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1.

1. Sử dụng giải thuật FIFO. Số lượng lỗi trang và số lần thay trang sẽ là:

- a) 18 - 15
- b) 17 - 17
- c) 18 - 18
- d) 17 - 14

2. Sử dụng giải thuật OPT. Số lượng lỗi trang và số lần thay trang sẽ là:

- a) 13 - 10
- b) 17 - 17
- c) 18 - 18
- d) 18 - 15

3. Sử dụng giải thuật LRU. Số lượng lỗi trang và số lần thay trang sẽ là:

- a) 18 - 18
- b) 18 - 15
- c) 17 - 14
- d) 13 - 13

4. Thrashing xảy ra khi:

- a) Tổng lượng bộ nhớ trong working-set lớn hơn bộ nhớ vật lý.
- b) Thời gian một quá trình đợi cho việc thay trang nhiều hơn thời gian thực thi.
- c) 2 câu trên đều đúng.
- d) 2 câu trên đều sai.

5. Chọn phát biểu sai:

- a) Kích thước của một trang là cố định do cần sự đồng bộ giữa các thiết bị phần cứng và phần mềm.
- b) Vùng nhớ cấp phát cho kernel phải liên tục.
- c) working set phản ánh program locality.
- d) Bộ nhớ ảo có kích thước lớn hơn bộ nhớ vật lý rất nhiều lần.

6. Chọn phát biểu sai về việc cấp phát vùng nhớ cho kernel:

- a) Cấp phát kiểu buddy sẽ có hiện tượng phân mảnh dữ liệu.
- b) Bộ cấp phát sẽ ưu tiên chọn trong slab empty trước nhất.**
- c) Các buddy có thể kết hợp với nhau tạo thành phân đoạn lớn hơn.
- d) Khi không còn slab empty bộ cấp phát sẽ thay những đối tượng used tương tự như việc thay trang.

Giả sử thời gian truy cập vùng nhớ ma = 10ns, tỷ lệ xảy ra lỗi trang p là 0.02, sử dụng HDD có thời gian chuyển trang là 5ms.

7. Thời gian truy xuất hiệu quả effective access time là:

- a) 100010 ns.
- b) 4900000 ns.
- c) 10980 ns.
- d) 100000,2 ns

8. Tỷ lệ so với hiệu suất lý tưởng:

- a) 0.0001.
- b) 0.05.
- c) 0.0009.
- d) 0.02.